XTOS™

Pattern User Manual

nptest®

Copyright Notice

This document and all information contained herein is proprietary to NPTest and protected by copyright or other laws of the United States and other countries. No part of this publication may be distributed to another party, reproduced, or copied without the express permission of NPTest. Additional copies of this, and any other publication, are available at any NPTest sales office.

Accelerated Vector Mode, AVM, APTEON, ASAP, BatchProbe, DeFT, IDS, IDS OptiFIB, ISOCHRONOUS FABRIC INTERFACE, ITS, IVS, Keep Alive, NanoBlade, NPower, NxGenConnect, NxGenSupport, OptiCA, SABER, Sapphire NP, Sequencer Per Pin, TimeStamper, TrueEdge, Ultra Low Voltage, Voice Grade, XTOS, and Yosemite are trademarks or registered trademarks of NPTest.

The following are trademarks or registered trademarks of their respective companies or organizations:

UNIX / X/Open Company Ltd.

Sun Microsystems, Sun Workstation, OpenWindows, SunOS, NFS, Sun-4, SPARC, SPARCstation, Java, Solaris / Sun Microsystems.

Ethernet / Xerox Corporation

Microsoft, Windows, Windows NT / Microsoft Corporation

IEEE / Institute of Electrical and Electronics Engineers, Inc.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

A reader comment form is provided at the end of this publication. Suggestions and comments are appreciated.

In no event shall NPTest be liable for any injury or loss, whether direct or indirect, or consequential damages of any kind caused by a failure to adhere to the safety standards and procedures in this manual.

© 2003 NPTest. All rights reserved.

P/N 57010495 Rev 6
ECO Number–27668

Printed in U.S.A.
December 2003

Contents

Preface ..... **ix**
Intended Audience ix
Revision History ix
Related Publications ix

Chapter 1. Pattern File (STIL) Description **1-1**
1.1 XTOS Pattern File (STIL) Structure 1-1
1.2 STIL Blocks 1-3
1.3 STIL Statement 1-3
1.4 Header Statement 1-3
1.5 Include Statement 1-4
1.6 Signals Block 1-4
1.7 SignalGroups Block 1-5
1.7.1 F2 and Fn Mode Declaration 1-6
1.8 ScanStructures Block 1-6
1.8.1 Scan Data Alignment—Scan Padding 1-7
1.9 Timing and WaveformTable Blocks 1-7
1.9.1 Resource Tags 1-8
1.9.2 Event States in STIL 1-9
1.9.3 Waveform Character 1-10
1.9.4 Merge Waveforms 1-10
1.10 PatternBurst and PatternExec Blocks 1-12
1.11 Procedures 1-13
1.12 MacroDefs 1-14
1.13 Pattern Block 1-14
1.13.1 WaveformTable Reference 1-15
1.13.2 Vector Statement 1-16
1.13.3 Condition Statement 1-16
1.13.4 Call Statement 1-16
1.13.5 Macro Statement 1-17
1.13.6 Loop Statement 1-18
1.13.7 MatchLoop Statement 1-18
1.14 Pattern Examples 1-19
1.14.1 Example 1 1-19
1.14.2 Example 2 1-20

1.15 Additional STIL Statements ... 1-22
1.15.1 Annotations ... 1-22
1.15.2 Comments ... 1-22
1.15.3 UserKeywords ... 1-22
1.15.4 Label Statement ... 1-22

Chapter 2. Hardware Support ... 2-1
2.1 Subroutine Memory Support ... 2-1
2.2 Sync and Trigger Support ... 2-4
2.3 Continuous Call Application ... 2-5
2.4 Match Vector ... 2-7
2.5 Keep Alive Vector ... 2-8
2.6 Parameter Passing ... 2-9
2.7 DC Channel Pattern Application ... 2-10
2.7.1 Usage Model ... 2-10

Chapter 3. Scan Support ... 3-1
3.1 ScanIn and ScanOut Declaration ... 3-1
3.2 Shift Statement ... 3-1
3.3 ScanStructure Block ... 3-2
3.4 Scan Pattern Examples ... 3-2
3.4.1 Example 1 ... 3-2
3.4.2 Example 2 ... 3-5
3.4.3 Example 3 ... 3-8

Chapter 4. Analog Support ... 4-1
4.1 Digitized Analog Pattern Example ... 4-1

Chapter 5. XTOS Pattern Compiler and Composite Linker ... 5-1
5.1 Pattern Complier ... 5-1
5.1.1 XTOS Pattern Complier Command Line Interface—Windows 2000 ... 5-1
5.1.2 NameMap Support in XTOS Pattern Compiler ... 5-2
5.1.3 XTOS Pattern Complier Command Line Interface—Solaris ... 5-3
5.2 Pattern Decomplier ... 5-4
5.2.1 XTOS Pattern Decomplier Command Line Interface—Windows 2000 ... 5-5
5.2.2 XTOS Pattern Decomplier Command Line Interface—Solaris ... 5-5
5.3 XTOS Pattern Composite Linker ... 5-6
5.3.1 Compositing Patterns ... 5-8
5.3.2 XTOS Pattern CompositeLink Command Line Interface—Windows 2000 ... 5-9
5.3.3 XTOS Pattern CompositeLink Command Line Interface—Solaris ... 5-9

Chapter 6. Test Template for Pattern Loading ... 6-1

Index ... Index-1

Figures

Figure 1-1 Timing Block and STIL Pattern Connection 1-8

Tables

Table 1-1 Drive States 1-9
Table 1-2 Compare State 1-9
Table 1-3 Expect States 1-9
Table 1-4 Indeterminate States 1-10
Table 1-5 WFC Conventions 1-10
Table 2-1 Maximum Total Latency 2-4

Preface

This XTOS™ Pattern User manual describes the details of the source syntax accepted by the XTOS Pattern Compiler.

Intended Audience

This manual is intended for use by anyone using the Sapphire NP™ test system with XTOS software. Familiarity with the digital testing concepts, and prior test experience with the IEEE® 1450 Standard Test Interface Language (STIL) for Digital Test Vector Data Standard are recommended.

Revision History

The following table presents the revision history of this publication for the current revision and, if applicable, the previous two revisions.

Revision Number	Date	Notes
6	12/03	Enhanced Chapter 3 “Scan Support.”
5	12/03	Updated to clarify information in Chapter 1.
4	11/03	Updated to conform to latest documentation standards.

Related Publications

In addition to the IEEE standard 1450, these NPTest® publications provides additional information for some of the topics covered in this publication. References to these related publications are made in the text of this publication whenever appropriate.

XTOS Software Tools Manual, P/N 57510507

Pattern File (STIL) Description 1

This chapter describes the basic capabilities of the pattern source file. The pattern source file contains statements providing functional test vectors for the Sapphire NP test system. The pattern file provides a convenient way to specify the stimulus and expected response data for functional testing.

XTOS™ pattern language is based on IEEE® 1450 standard, Standard Test Interface Language (STIL) and its extensions. Some extensions are added through STIL's UserKeywords to expand the scope to support NPTest's hardware capabilities and mixed-signal applications.

1.1 XTOS Pattern File (STIL) Structure

The XTOS software facilitates the test pattern from data EDA-STIL phase to the test target phase through a series of phase steps as follows:

- A utility is provided to break-up the STIL file into various block (Signals, WavefromTable, and so on) related files for easier file management. This is a very useful tool in handling fairly large STIL files.
- Tester targeting:
  - A resource tag is added in the Timing block to allow timing and pattern sequence matching communication by mapping event lists to timing waveform sequences.
  - WaveformTable is the only one that needs to be modified to address the timing and pattern communication.
- Compile a single STIL pattern file:

  This phase compiles a source STIL file into a XTOS specific binary loadable image. For example, a pattern PAT1.stil file, after compiling, results in a PAT1.pof (pattern object file) directory containing the following three files:

  vpr — virtual pattern representation—Hardware independent intermediate output from the STIL parser.

rpr — runtime pattern repository—Stores pattern related mapping and meta-data generated by the LIG.

lpi — loadable pattern image—Binary image suitable for direct loading on the target instrument hardware.

These files are for use by XTOS software only.

- Generate a composite image for a group of patterns:

  This phase is used to optimize the pattern resources in Sapphire NP™ test systems. This is implemented during a post STIL compile step using an application called CompositeLink. The compositing of patterns provides an optimized scheme of resource usage by creating a common set for the following resources in a composite directory:

  – Common Event Sequence Store Memory (ESSM) locations

  – Common Source Data Select (SDS)

  – Common Triggers/Syncs

  – Common Subroutine Data

  Note that the set of patterns that can be processed into a single composite should belong to the same timing domain.

Details related to the process of compiling and compositing the STIL patterns are covered in the subsequent chapters of this document.

STIL maintains a define-before-use mandate on most information. This requirement sets a general ordering of the STIL blocks. There are also several exceptions to the define-before-use rule. In general, the Pattern blocks are located at the end of the STIL file, and after the PatternBurst block. The STIL's data block organization is:

- Signals
- SignalsGroups
- Timing
- PatternBurst
- PatternExec
- Pattern

XTOS pattern structure supports the following STIL top-level statements and features in processing the pattern data:

- STIL statement
- Header statement
- Include statement
- Signals statement
- SignalGroups statement
- ScanStructures statement

- Timing statement
- MacroDefs statement
- PatternBurst
- PatternExec
- Pattern statement
- UserKeywords
- Annotations

In addition to the newest features introduced in the STIL (1450.3) standard, a few extensions have been made to the standard to take advantage of the NPower™ architecture. Most of these extensions make use of the UserKeyword construct provided by STIL.

1.2 STIL Blocks

The XTOS pattern file structure supports STIL standard (1450-1999) syntax for direct compile of the STIL pattern from an ATPG tool. Additionally, mapping of the STIL names to the Sapphire NP environment through NameMap block (P1450.1) and the mapping of timing and pattern data (P1450.3) are also supported. Extensive usage of the STIL UserKeywords statement is also made to expand the STIL syntax to best utilize the architectural advantages of the Sapphire NP systems and mixed-signal applications.

A description of the STIL block statements supported in the XTOS pattern structure is listed in this following sections.

1.3 STIL Statement

A STIL pattern file must start with a STIL statement. The STIL statement may be either a single statement or a block statement, such as:

```
STIL 1.0;      // single statement to indicate 1450-1999 STIL file
STIL 1.0 {     // block to include P1450.3 extension
     TRC D0.8;
}
```

1.4 Header Statement

An optional block after the STIL statement is the Header block. The Header block must be the first block after the STIL statement and before any other STIL blocks in the file. Header blocks contains file-oriented information.

The defined statements of the Header block are:

```
Header {
```

```
        Title "title_string";Date "date_string"; // Title of the file
        Source "source_string"; // Date of the file being generated
        History {  } // How is this file generated
}
```

1.5 Include Statement

The STIL Include statement may appear any place a STIL statement/block may be found.

It is recommended to always quote filenames in the Include statement. Generally a filename contains at least a period (.) character, which has other semantic meanings in STIL (concatenation).

It is recommended that top level STIL files are named with the suffix .stil, while all the include files are named with other suffix, such as .inc.

The include file path is supported by both a relative file path and a absolute file path.

The IfNeed option is ignored in XTOS environment. The pattern compiler includes the Include file regardless of the IfNeed option.

The Include file must start with the STIL statement as the first statement in the file.

1.6 Signals Block

Only one Signals block is allowed for one STIL pattern file. The Signals block will not have a name associated to it.

This block defines the Signals used in the test program and their types (such as In, Out, InOut, Supply, and Pseudo). The type In, Out, or InOut is used to define a regular signal. The type Supply is used for power or ground pins, there is no functional data with these pins. The type Pseudo is used to support certain test operations with no device signal association (such as sync trigger pin). It also defines the base signal names for all the ports of the design.

Signal names must begin with an alphabetic character or an underscore. Signal names can also be enclosed in double quotes. Signal names with double quotes use the two double quotes as part of the signal name. In other words, signal "Abus" is a different signal from signal Abus. Signal names are also case sensitive and hence the same names with different cases refer to different signals.

The syntax of the Signals block is:

```
Signals {
  (sig_name <In | Out | InOut | Supply | Pseudo>;)*
}
```

Below is a Signals block statement example:

```
Signals {
        Abus_0          In;
        Abus_1          In;
```

```
        Bbus_0          Out;
        Bbus_1          Out;
        Cbus_0          InOut;
        Cbus_1          InOut;
        Dbus_0          InOut;
        Dbus_1          InOut;
}
```

1.7 SignalGroups Block

This block is an optional block and refers to a group of signals. These groups can only refer to a collection of existing signals. A SignalGroup of one signal simply results in the renaming of that signal. This feature facilitates a mechanism to reference the same signal by any number of alias names. There are no restrictions on the types of the signals that may be grouped together. However, a particular signal can appear only once in a group. Unlike the Signals statement, multiple SignalGroups statements are allowed in a test pattern as long as these SignalGroups are assigned different names. Only one *global* SignalGroups (one without any name) is allowed per test pattern. The grouping of signal group should use single quotation marks ('). The default order of signals in any signal group is from left to right. For example: group = '"A"+"B"+"C"'; then the data of group = XYZ will map the assignment as "A"=X, "B"=Y, "C"=Z.

The syntax of the SignalGroups block is:

```
SignalGroups {
  (group_name = 'signal_reference_expression' ;)*
  (group_name = 'signal_reference_expression' {
       (DataBitCount decimal_integer;)
   })*
}
```

Where:

signal_reference_expression
: Contains the signal name or signal group name with expression operators, such as + and -. STIL does not allow duplicate signal names in the same group, so parenthesis may also be used as part of expression to eliminate a duplicate signal before evaluation.

DataBitCount
: Indicates number of bits of data for multiple-bit data.

Below is a SignalGroups example:

```
SignalGroups {
       Abus = 'Abus_1 + Abus_0';
        Bbus = 'Bbus_1 + Bbus_0';
        Cbus = 'Cbus_1 + Cbus_0';
        Dbus = 'Dbus_1 + Dbus_0';
        allPins = 'Abus + Bbus + Cbus + Dbus';
}
```

1.7.1 F2 and Fn Mode Declaration

If some signals or signal groups need to have multiple F-data to support the data rate, then the attribute of DataBitCount can be used to imply the number of F-data bits. The DataBitCount is number of signals * number of F-data bits. For example, F2 mode with signal group of 2 signals will have DataBitCount = 4. F4 mode with signal group of 2 signals will have DataBitCount = 8.

It should be noted that a databit count is the number of signals multiplied by the number of F-data bits only when defining the databit count inside a SignalGroups. It is also valid to declare a databit count for a single signal.

The DataBitCount does not represent a direct declaration of F-data bits. Instead the maximum DataBitCount is determined for each signal across the whole pattern (considering each occurrence in the in the Signal or SignalGroup block). The remaining F-data bits are also padded, if the signal expression calls for fewer fdata bits than the maximum.

Below is an example of F2 mode SignalGroups of Abus and Bbus:

```
SignalGroups {
        Abus = 'Abus_1 + Abus_0' {DataBitCount 4;}
        Bbus = 'Bbus_1 + Bbus_0' {DataBitCount 4;}
        Cbus = 'Cbus_1 + Cbus_0';
        Dbus = 'Dbus_1 + Dbus_0';
        allPins = 'Abus + Bbus + Cbus + Dbus';
}
```

Below is an example of F4 mode SignalGroups of Abus and F2 mode for Bbus:

```
SignalGroups {
        Abus = 'Abus_1 + Abus_0' {DataBitCount 8;}
        Bbus = 'Bbus_1 + Bbus_0' {DataBitCount 4;}
        Cbus = 'Cbus_1 + Cbus_0';
        Dbus = 'Dbus_1 + Dbus_0';
        allPins = 'Abus + Bbus + Cbus + Dbus';
}
```

The D-4032 instrument supports up to F4 data rate.

1.8 ScanStructures Block

This block is an optional block and describes the interconnection of scan chain information in the DUT. Details related to the length of the scan chain, definitions of the scan input and output signal associated with the chain and the definition of the internal scan cells are contained in this block.

The syntax of the ScanStructures block is:

```
ScanStructures scan_name {
    (ScanChain chain_name {
        ScanLength decimal_integer;
        (ScanOutLength decimal_integer;)
        (ScanCells cell_name_list;)
        (ScanIn signal_name;)
        (ScanOut signal_name;)
        (ScanMasterClock signal_name_list;)
        (ScanSlaveClock signal_name_list;)
        (ScanInversion <0 | 1>;)
    })*
}
```

1.8.1 Scan Data Alignment—Scan Padding

When a scan pattern has more than one scan chain, and the scan chains are not all the same length, then the XTOS pattern compiler will provide padding to make all scan chains equal length. The scan length of each scan chain is specified in the ScanLength and ScanOutLength statements in the ScanChain block.

The XTOS pattern compiler uses the longest scan chain to determine the padding required for the shorter ones. The pad data is added to the beginning of the scan-in data and to the end of scan-out data.

1.9 Timing and WaveformTable Blocks

Timing and WaveformTable blocks are required in a STIL pattern to define timing information. The Timing block contains definitions of event behavior across all signals in the test pattern. A collection of these definitions is contained in the WavefromTable blocks.

All WaveformTables used in a pattern must be defined in one Timing block. WaveformTable block consists of one or more Waveform(s) for each signal or signal group. The Waveform statement is used to map a signal's waveform character (WFC) to an event list, also known as the timing sequence.

The syntax of the Timing block is:

```
Timing timing_domain_name {
    (SignalGroups group_name;)
    (WaveformTable Waveform_table_name {
        (<period_name>) Period time_expression;
        Waveforms {
            (signal_ref {
                (<sequence_name>) WFC_List {
                    ((time_expression) (event);)*
                })*
            })*
        }
    })*
}
```

Where

<period_name>,<sequence_name>
: The <resource tag> to map pattern's Waveform to test program's timing sequence see “Resource Tags”.

event
: STIL's event state see “Event States in STIL” on page 1-9.

The relationship between the period name or sequence name of a test program's Timing block and resource tag in a STIL pattern is shown in Figure 1-1.

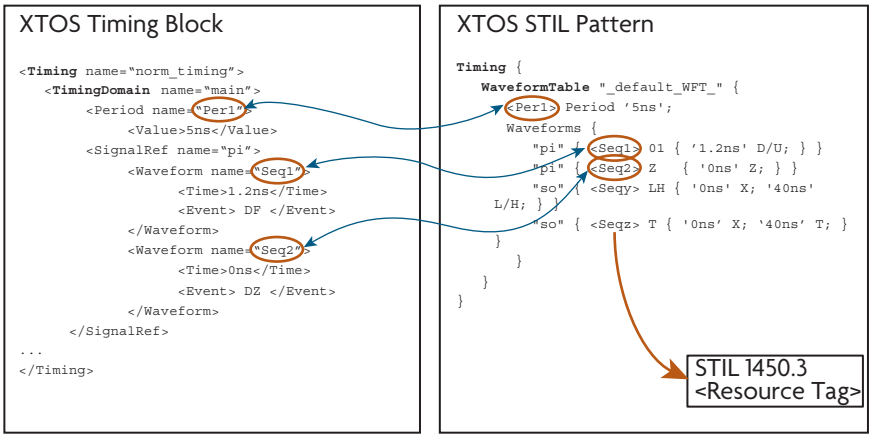

Figure 1-1: Timing Block and STIL Pattern Connection

1.9.1 Resource Tags

As per the P1450.3 extension, XTOS software uses the resource tag (<period_name>, <sequence name>) to map between the waveform character (WFC) and the timing sequence name in the test program. The <period_name> and <sequence_name> must be the timing sequence for the corresponding signal in the Timing block of the test program. If these names are not found in the corresponding Timing block, then timing shaping error will happen after the pattern is installed.

1.9.2 Event States in STIL

STIL has a total of 23 event states. They have fixed value and purpose. Table 1-1 through Table 1-4 list the available event states.

Table 1-1: Drive States

Drive State	Description	Timing Events
D	ForceDown	D0
U	ForceUp	D1
Z	ForceOff	DZ
P	ForcePrior	NOP
N	ForceUnknown	DZ

Table 1-2: Compare State

Compare State	Description	Timing Events
L	CompareLow	T0
H	CompareHigh	T1
X or x	CompareUnknown	X
T	CompareOff	TZ
V	CompareValid	X
l	CompareLowWindow	T0
h	CompareHighWindow	T1
t	CompareOffWindow	TZ
v	CompareValidWindow	X

Table 1-3: Expect States

Expect State	Description	Timing Events
R	ExpectLow	X
G	ExpectHigh	X
Q	ExpectOff	X
M	Marker	X

Table 1-4: Indeterminate States

Indeterminate State	Description	Maps to
A	LogicLow	DZ
B	LogicHigh	DZ
F	LogicZ	DZ
?	Unknown	DZ

1.9.3 Waveform Character

The waveform character (WFC) is a user defined ASCII character representing a timing sequence. The waveform character can be one of 63 characters (upper-case and lower-case alphabetic, numeric, and underscore). The general convention to use the WFC and representations are listed in Table 1-5.

Table 1-5: WFC Conventions

WFC	Conventional Waveform Character Representation
0	Drive low
1	Drive high
Z	Disable the driver, Drive to tri-state
H	Compare high
L	Compare low
X	Do not compare

1.9.4 Merge Waveforms

STIL supports a single WFC to map to a single event in a event list, such as:

```
Abus    { 0 { '0ns' D;} }
Abus    { 1 { '0ns' U;} }
```

STIL also supports merge waveforms that map two or more WFCs to multiple event states in a event list, such as:

```
Abus    { 01 { '0ns' D/U;} }
Bbus    { LH { '0ns' L/H;} }
Cpin    { aA { '0ns' D/U;} }
```

The XTOS pattern compiler takes the merge waveform and implies F-data and FM-mode operation. The combined D/U or L/H event state implies that the current waveform (timing sequence) has F-data. The combined L/H/X, L/X, or H/X event state implies that the corresponding signal has FM mode, such as:

```
Bbus    { LHX { '0ns' L/H/X;} }
```

The WaveformTable can be defined as a single waveform per signal/signal group. In this construct, the selection of WaveformTable in each vector has a fixed WFC selection and fixed timing sequence to map with the vector.

```
Timing "Vec_Def" {
    WaveformTable VecDef_0{
        <period_0> Period '10ns';
        Waveforms {
            Abus        { <Abus_D> 01 { '0ns' D/U;} }
            Bbus        { <Bbus_S> LH { '25ns' L/H;} }
            Cbus        { <Cbus_D> 01 { '0ns' D/U;} }
            Dbus        { <Dbus_S> LH { '25ns' L/H;} }
        }
    }
    WaveformTable VecDef_1{
        <period_0> Period '10ns';
        Waveforms {
            Abus        { <Abus_S> LH { '25ns' L/H;} }
            Bbus        { <Bbus_D> 01 { '0ns' D/U;} }
            Cbus        { <Cbus_S> LH { '25ns' L/H;} }
            Dbus        { <Dbus_D> 01 { '0ns' D/U;}
        }
    }
    WaveformTable VecDef_2{
        <period_0> Period '10ns';
        Waveforms {
            Abus        { <Abus_Z> Z  { '0ns' Z;} }
            Bbus        { <Bbus_S> LH { '25ns' L/H;} }
            Cbus        { <Cbus_Z> Z  { '0ns' Z;} }
            Dbus        { <Dbus_S> LH { '25ns' L/H;} }
        }
    }
    WaveformTable VecDef_3{
        <period_0> Period '10ns';
        Waveforms {
            Abus        { <Abus_S> LH { '25ns' L/H;} }
            Bbus        { <Bbus_Z> Z  { '0ns' Z;} }
            Cbus        { <Cbus_S> LH { '25ns' L/H;} }
            Dbus        { <Dbus_Z> Z  { '0ns' Z;} }
        }
    }
}
```

The WaveformTable also can be defined as all waveform events are defined for each signal/signal group. In this construct, the selection of WaveformTable in each vector has a set of WFC selection and the selection of WFC maps to the corresponding timing for that vector.

```
Timing "Vec_Def" {
    WaveformTable VecDef {
        <period_0> Period '10ns';
        Waveforms {
            Abus    { <Abus_D> 01 { '0ns' D/U;}
                      <Abus_Z> Z  { '0ns' Z;}
                      <Abus_S> LH { '25ns' L/H;} }
            Bbus    { <Bbus_D> 01 { '0ns' D/U;}
                      <Bbus_Z> Z  { '0ns' Z;}
                      <Bbus_S> LH { '25ns' L/H;} }
            Cbus    { <Cbus_D> 01 { '0ns' D/U;}
                      <Cbus_Z> Z  { '0ns' Z;}
                      <Cbus_S> LH { '25ns' L/H;} }
            Dbus    { <Dbus_D> 01 { '0ns' D/U;}
                      <Dbus_Z> Z  { '0ns' Z;}
                      <Dbus_S> LH { '25ns' L/H;} }
        }
    }
}
```

If F2 data is used for some of the signals, then the representation of multiple data waveform is needed. The bracketed event [0] represents the first F-data and the bracketed event [1] represents the second F-data.

```
Timing "Vec_Def" {
    WaveformTable VecDef_0{
        <period_0> Period '10ns';
        Waveforms {
            Abus    { <Abus_DF2> aA { '0ns' D/U[0]; '50ns' D/U[1];}
                      <Abus_Z>    Z { '0ns' Z;}}
            Bbus    { <Bbus_SF2> bB { '25ns' L/H[0]; '75ns' L/H[1];}
                      <Bbus_X>    X { '0ns' X;}}
            Cbus    { <Cbus_D>   01 { '0ns' D/U;}
                      <Cbus_Z>    Z { '0ns' Z;}}
            Dbus    { <Dbus_S>   LH { '25ns' L/H;}
        }
    }
}
```

1.10 PatternBurst and PatternExec Blocks

These two blocks are required blocks in the XTOS pattern files. They ensure that the PatternExec block contains the name of the PatternBurst block and that the PatList contains the name of the Pattern block. XTOS software uses functional equivalent constructs known as functional test templates and the PatternBurst block from the test program in place of these blocks.

The syntax of the PatternBurst is:

```
PatternBurst pat_burst_name {
    PatList {
        Pattern_or_burst_name;
    }
}
```

The syntax of the PatternExec is:

```
PatternExec pat_exec_name {
    Timing timing_name;
    PatterbBurst pat_burst_name;
}
```

1.11 Procedures

The STIL Procedures block is an optional block. A single unnamed Procedures block may exist and all the procedures are defined in the Procedures block. XTOS's subroutine capability is also supported through the STIL's procedure with the NPTSubroutine UserKeyword. This UserKeyword identifies it is a subroutine instead of a regular STIL procedure.

The syntax of the Procedures block is:

```
Procedures (optional_Procedure_block_name) {
    (( NPTSubroutine { procedure_name;} )
     procedure_name {
         (pattern_statement;)*
     })*
}
```

Each procedure contains any combination of the STIL pattern statements, such as Vector, Condition, and WaveformTable reference. In addition to these regular statements, a procedure may also contain the Shift statement for scan vector.

Procedure is invoked through a Call statement. When a procedure is called, no information about the current signal state is transferred into that procedure. A WFC for all signals used in the procedure must be specified, and a WaveformTable reference must also be defined by the first Vector in the procedure. When a procedure ends, the context of the last statement of the procedure will not transfer back to its caller. A procedure does not return any change back to the calling environment no matter how the signals may have changed during the procedure. The Vector after the Call statement will inherit the states from the Vector prior to the Call statement. Any STIL Procedures, without a NPTSubroutine attached, are expanded as in-lined vectors in the system's Main memory section.

This is a example of the correct Procedure with WaveformTable references and complete signal state assignments:

```
Procedure {
    "Proc_A" {
```

```
        W "timing_X";
        V { "one" = 1; "two" = L; }
        V { "two" = H; }
        V {  }
    }
}
```

Below is an improper Procedure, it is missing WaveformTable references and has incomplete signal state assignments:

```
Procedure {
    "Proc_A" {
        V { "one" = 1; }
        V { "two" = H; }
        V {  }
    }
}
```

1.12 MacroDefs

The STIL MacroDefs block is an optional block.

The syntax of the MacroDefs block is:

```
MacroDefs (optional_macrodef_block_name) {
     macro_name {
         (pattern_statement;)*
     })*
}
```

Each macro contains any combination of the STIL pattern statements, such as Vector, Condition, and WaveformTable reference.

Macro is invoked through a Macro statement. When a macro is called, it maintains the state of the calling environment. Only changes to the signal states, (such as different WFC for signals or different WaveformTable reference), need to be specified in the macro. A macro returns all changes back to the calling. The Vector after the Call statement inherits the states from the macro.

```
Macro {
    "Macro_A" {
        V { "one" = 1; }
        V { "two" = H; }
        V {  }
    }
}
```

1.13 Pattern Block

This Pattern block is the main section of a STIL pattern. This block defines the contents of the pattern vectors and contains one or more of the following Pattern statements:

WaveformTable reference
: W *waveformTable_name* defines the waveform table to use for the Vector translation.

Vector
: V { } defines stimulus and response for one test vector.

Condition
: C { } defines a stimulus and/or response to be set up, but deferred from being performed until a Vector statement is defined.

Call
: Call *procedure_name*—Where procedure is a previously defined Procedures block section. XTOS replaces the Call statement by in-lining the vectors in the procedure for a regular procedure. If the procedure is a NPTSubroutine procedure, then the Call is converted to a Call opcode.

Macro
: Macro *macro_name*—Initiates the previously defined MacroDefs block section. XTOS replaces the Macro statement by in-lining the vectors in the Macro_Defs block.

Loop
: Loop *n*—Where *n* is an integer number of times on which a block of pattern statements is looped.

MatchLoop
: MatchLoop *n*—Where *n* is an integer number of times on which a block of pattern statements is looped until all pattern statements are executed without differences between the output information in the pattern statement and the response from the device or until *n* number of loops is executed.

GoTo
: Goto *Label*—Currently XTOS software does not support this statement.

BreakPoint statement
: Defines a location in the pattern where the device is in a stable state. XTOS software currently processes this statement as a NULL vector.

1.13.1 WaveformTable Reference

A WaveformTable Reference statement must be present before any Vector statements. This statement is used to identify the timing sequences associated with the WFCs assigned to signals in the vectors following this statement. A WaveformTable Reference statement must precede the first vector in the section. The selected WaveformTable remains in force until another WaveformTable Reference statement is used to select a different WaveformTable.

The two possible syntax of the WaveformTable reference are:

```
WaveformTable wavefromtable_name;
W waveformtable_name;
```

1.13.2 Vector Statement

The Vector statement defines stimulus and response for one test vector. The first Vector statement in a Pattern block needs to have a WFC assigned for every signal used in the pattern. After the first vector, subsequent vectors only need to contain signal assignment when there is a change to a different state (WFC) in that vector. Prior to the very first Vector statement, there must be the WaveformTable reference statement to specify the WaveformTable. If a Vector statement has empty content, it implies that every signal keeps the same WFC as the previous assignment.

The possible syntax of the Vector statement are:

```
Vector { (signal_assignment)* }
V      { (signal_assignment)* }
```

Where:

signal_assignment has syntax as signal_ref = WFC or WFC_list;

1.13.3 Condition Statement

The Condition statement does not generate actual test events. It only establishes new waveform assignment to signals that may be applied to the subsequent Vector statements until a change in WFC occurs.

The possible syntax of the Condition statement are:

```
Condition { (signal_assignment)* }
C         { (signal_assignment)* }
```

where:

signal_assignment has syntax as signal_ref = WFC or WFC_list;

Here is a Condition statement example:

```
Pattern XXY {
    W "default_waveformtable";
    C { all_signals = 0101000111XXXXX11001; }
    V { }             // first vector; value defined in C { }
    V { }             // first vector; value defined in C { }
V { Abus = HH; }
    }
}
```

1.13.4 Call Statement

A Procedure is invoked in the pattern section through the Call statement. STIL's Procedure supports parameter passing that the Call statement can pass the value assignment to the corresponding procedure. The Call statement does not generate a physical test cycle and all the signal assignments are used for parameter passing. The NPTest Subroutine version of Procedure does not allow a signal assignment statement in the Call statement.

The syntax of the Call statement is:

Call { (*signal_assignment***)* }**

Where:

signal_assignment has syntax as signal_ref = WFC or WFC_list;

Below is an example of a Call statement:

```
Procedure {
    "Proc_A" {
        W "timing_X";
        V { "one" = 1; "two" = L; }
        V { "one" = #; "two" = %; }
        V { "one" = 0; "two" = H; }
    }
}

Pattern XXY {
    W "timing_X";
    V { "one" = 0; "two" = H; }
    Call "Proc_A" { "one" = 1; "two" = H; }
    V { "one" = 0; }
    Call "Proc_A" { "one" = 0; "two" = L; }
    }
}
```

1.13.5 Macro Statement

A MacroDef is invoked in the pattern section through the Macro statement. STIL's macro supports parameter passing as in the Procedure. The Macro statement does not generate a physical test cycle and all the signal assignments are used for parameter passing.

The syntax of the Macro statement is:

Macro { (*signal_assignment***)* }**

Where:

signal_assignment has syntax as signal_ref = WFC or WFC_list;

Below is an example of a Macro statement:

```
MacroDefs {
    "Macro_A" {
        V { "one" = 1; }
        V { "one" = #; "two" = %; }
        V { "one" = 0; "two" = H; }
    }
}

Pattern XXY {
    W "timing_X";
    V { "one" = 0; "two" = H; }
```

```
        Macro "Macro_A" { "one" = 1; "two" = H; }
        V { "one" = 0; }
        Macro "Macro_A" { "one" = 0; "two" = L; }
        }
}
```

1.13.6 Loop Statement

STIL supports a repeating construct with Loop statement. A Loop statement maps a single vector into one or more RPT sequencer opcodes (RPT operation) or multiple Vector statements into a loop (BNZ operation). XTOS supports RPT in both Main and Subroutine memory, but only BNZ loop operation in the Subroutine memory. The multiple vector Loop construct of a STIL pattern is expanded to straight line vectors in the Main pattern section. Pattern compiler also takes care of the loop count, if the value is beyond the maximum count value that the hardware supports. In other words, the compiler may break a single Loop statement into multiple RPT vectors in order to to accommodate the pattern requirements given the test system limits. Pattern Compiler also takes care of the hardware restriction that does not allow back-to-back RPT opcode in two consecutive vectors. Pattern compiler ensures there is an INC opcode inserted between the two RPT opcode vectors and keeps the total test count identical to the source code cycle count.

The syntax of the Loop statement is:

Loop *loop_count* **{** (*vector_statement*)***** **}**

Below is an example of a loop in Match memory:

```
Pattern XXY {
    W "default_waveformtable";
    V { all_signals = 0101000111XXXXX11001; }
    Loop 10 { V { } }    // Simple Repeat 10 times
    Loop 33 {            // Will be expanded to straight line
         V { Abus = HH; }
         V { Bbus = LH; }
    }
}
```

1.13.7 MatchLoop Statement

The MatchLoop construct in STIL provides basic capability to wait for certain conditions (state of specific signal(s) or other condition) to be met before exiting the loop. It is used to support Match, Continuous, Sync, and Trigger capabilities in the system. The XTOS pattern compiler ignores the BreakPoint block in the MatchLoop construct. Some UserKeywords are added to support Sapphire NP hardware capabilities.

The syntax of the MatchLoop statement are:

```
MatchLoop < loop_count | Infinite > {
      (vector_statement)+
      BreakPoint { (vector_statement)+ }
}
```

1.14 Pattern Examples

The contents of PatternBurst, PatternExec and Pattern blocks are shown in the working examples below. These examples illustrate various STIL statements and their applications. The Signals, SignalGroups, Timing, and WaveformTable blocks are used through the application of the Include statement.

1.14.1 Example 1

A pattern that uses 1 bit of functional data per pin. Application of W, V, and Loop statements are illustrated here.

```
STIL 1.0;

Include "pattern_include.inc"; // Using Include file to hold identical
information
PatternBurst "dummy" {
    PatList { "Pass_all"; }
}

PatternExec {
    Timing "Vec_Def";
    PatternBurst "dummy";
}

Pattern "Pass_all" {
    W Vec_0;
    V {Abus=00; Bbus=LL;}
    V {         Bbus=LH;} // Abus still has value 00
    V {Abus=10;         } // Bbus holds value of LH
    Loop 1000 { // Repeat 1000 times
        V {Abus=11; Bbus=HH;}
    }
    V {Abus=Aa; Bbus=Bb;}
    V {Abus=aa; Bbus=bb;}

    W Vec_1; // Change Waveform table from Vec-0 to Vec-1
    V {Bbus=00; Abus=LL;}
    V {         Abus=LH;} // Bbus holds value of 00
    V {Bbus=10; Abus=HL;}
    V {Bbus=11;         } // Abus holds value of HL
    }
```

The Include file, pattern_include.inc, for the above Pattern is:

```
STIL 1.0 { TRC D08; }

Signals {
    Abus_0            InOut;
    Abus_1            InOut;
    Bbus_0            InOut;
    Bbus_1            InOut;
}

SignalGroups {
    Abus = 'Abus_1 + Abus_0'; // Abus_1 is MSB. Abus_0 is LSB
    Bbus = 'Bbus_1 + Bbus_0';
    allPins = 'Abus + Bbus'; // Abus_1 is MSB, Bbus_0 is LSB
}

Timing "Vec_Def" {
    WaveformTable Vec_0 {
        <period_0> Period '10ns';
        Waveforms {
            Abus  { <Abus_D>  01 { '0ns' D/U;}
                    <Abus_D2> aA { '0ns' D/U; '50ns' U/D;}
           }
            Bbus  { <Bbus_S>  LH { '25ns' L/H;}
                    <Bbus_S2> bB { '25ns' L/H; '75ns' H/L;}
                  }
        }
    }
    WaveformTable Vec_1{
        <period_1> Period '10ns';
        Waveforms {
            Abus  { <Abus_S> LH { '25ns' L/H;} }
            Bbus  { <Bbus_D> 01 { '0ns' D/U;} }
        }
    }
}
```

1.14.2 Example 2

A pattern that uses 2 bits of functional data per pin (F2 pin mode). Application of W, V, and Loop statements is illustrated here.

```
STIL 1.0;

// This is MyPat_F2.stil file

Include "MyPat_F2_include.inc";

PatternBurst "dummy" {
    PatList { "MyPat_F2"; }
}
```

```
PatternExec {
    Timing "Vec_Def";
    PatternBurst "dummy";
}

Pattern "My_pat_F2" {
    W Vec_0;
    //  FF FF           FF FF
    //  12 12           12 12
    V {Abus=00 00; Bbus=LL LL; }
    V {   }
    V {Abus=01 01; }
    Loop 500 {
        V {Abus=11 01; Bbus=HH LL; }}

    W Vec_1;
    V {Abus=LL LL; Bbus=00 00; }
    V {Abus=LH LH; }
    }
```

The contents of MyPat_F2_include.inc file are:

```
STIL 1.0 {
TRC D 0.8;
}

Signals {
       Abus_0         In;
       Abus_1         In;
       Bbus_0         Out;
       Bbus_1         Out;
}

SignalGroups {
       Abus = 'Abus_1 + Abus_0' {DataBitCount 4;}
       Bbus = ' Bbus_1 + Bbus_0' {DataBitCount 4;}
       allPins = 'Abus + Bbus';
}

Timing "Vec_Def" {
    WaveformTable Vec_0 {
        <period_0> Period '10ns';
        Waveforms {
            Abus  { <Abus_D> 01 { '0ns' D/U[0];  '50ns' D/U[1];}}
            Bbus  { <Bbus_S> LH { '25ns' L/H[0]; '75ns' H/L[1];}}
        }
    }
    WaveformTable Vec_1{
        <period_1> Period '10ns';
        Waveforms {
            Abus  { <Abus_S> LH { '25ns' L/H[0]; '75ns' H/L[1]; }}
            Bbus  { <Bbus_D> 01 { '0ns' D/U[0];  '50ns' D/U[1];} }
        }
    }
```

```
}
```

1.15 Additional STIL Statements

STIL supports two forms of comment information as part of STIL statements:

- Annotations
- Comments

1.15.1 Annotations

Ann is the STIL keyword, followed by special identifies to enclose the annotation body.

The syntax of the Annotation statement is:

```
Ann {* text string for annotation *}
```

Annotations are maintained through the XTOS system and preserved as comments. Generally, to maintain association of an Ann statement as a part of STIL, the Ann statement should be placed inside a block construct.

1.15.2 Comments

STIL supports the second form of comment information with Comment statements.

The syntax of the Comment statement is:

```
// comment text string until end of line
/* comment text string in between */
```

Comment statements are not preserved in XTOS. These statement are converted to Ann statements, which can be preserved in XTOS. This is a user controlled option.

1.15.3 UserKeywords

STIL supports syntax extensions through the UserKeywords statement. XTOS uses the UserKeywords to define various NPTxxxx UserKeywords to support Sapphire NP hardware capabilities and mixed signal applications. All XTOS UserKeywords have a NPT prefix.

1.15.4 Label Statement

Any Pattern statement can have an optional associated label to identify start/stop vectors or a specific pattern location (scan pattern set).

```
Pattern_label :
```

Labels are scoped to its Pattern block, and to individual Procedure bodies. Labels are required to be unique. Any label within Macro is ignored.

Hardware Support 2

The XTOS pattern syntax also supports Sapphire NP's hardware capabilities and mixed signal applications. The following sections cover these extensions.

2.1 Subroutine Memory Support

Sapphire NP test system has subroutine memory as one of the data sources for pattern data. STIL has a Procedure construct in the 1450-1999 standard however, not all the procedures can be mapped into subroutine usage due to:

- Subroutine is a limited resource in the test system. STIL's procedure/macro is a software construct that may be used extensively. For example, scan vectors in STIL are 100% in procedure/macro construct due to the nature of scan load/unload/capture vectors.
- Sapphire NP's subroutine memory can not handle parameter passing (no equivalent PARAM opcode available).
- Sapphire NP subroutine memory size is 4k.

To preserve the capability of digital STIL patterns (especially scan patterns) to be processed by EDA ATPG tools for failure analysis, XTOS keeps the Subroutine extension syntax as that of a STIL procedure statement.

A STIL UserKeyword is defined to support Sapphire NP’s subroutine memory. However, the STIL procedure statement is used as much as possible as the base of Subroutine syntax. If the Sapphire NP subroutine capability is not used, the default behavior for the procedure and macro statements is inline with the pattern statements. Additionally, a UserKeyword is defined to support hardware capability, in which case an EDA tool may not be able to simulate this pattern directly from STIL syntax.

If a specific STIL procedure is targeted to be a Subroutine, then the keyword NPTSubroutine needs to be attached in front of the procedure as an identifier. If there is any Loop construct or load counter in the Subroutine, then NPTSelectCounter should be used to select one of the counters as the loop controller. XTOS pattern compiler

handles the proper loop expansion to satisfy the hardware requirement. The test system has eight counters available for loop control, A through H, and the last two pair E/F and G/H can be concatenated.

The extension syntax for Subroutine support is:

```
UserKeywords NPTSubroutine NPTOpcode NPTSelectCounter ;
NPTSubroutine {subroutine_name;}
NPTOpcode {CALLZ | LDC | BNZ (Operand);}
```

Where:

1. NPTSubroutine is used to declare the following STIL Procedure block is a Subroutine.
2. NPTOpcode is used either in main pattern section as CALLZ, or LDC opcodes, or in the subroutine section as LDC, or BNZ opcodes.
3. *subroutine_name* is the subroutine name.
4. NPTSelectCounter is used to select a counter (A through H).
5. Operand is a count (for LDC) or label (for BNZ)

Example 1: EDA Compatible Subroutine Statements:

```
UserKeywords NPTSubroutine NPTOpcode NPTSelectCounter;
...
Procedures {
    PROC_STIL {
        WaveformTable MY_WFT;
        Vector {sig1=##; sig2=11;}
        Vector {sig1=00;}
    }
    NPTSubroutine "MY_SUBR"; // EDA readable
    MY_SUBR {
        WaveformTable MY_WFT;
        Vector {sig1=00; sig2=11;}
        Vector {sig1=11;}
    }
    NPTSubroutine "MY_SUBR2"; // EDA readable
    MY_SUBR2 {
        WaveformTable MY_WFT;
        Vector {sig1=00; sig2=11;}
        MY_LOOP;
        NPTSelectCounter A; // Select Counter A
        Loop 95 {
            Vector {sig1=10; sig2=10;}
            Vector {sig1=00; sig2=01;}
            Vector {sig1=11;}
        }
    }
}

Pattern TRY {
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=11;}
```

```
    Vector {}
    Call PROC_STIL {sig1=11;}
    ...
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=00;}
    Vector {sig1=01; sig2=10;}
    Call MY_SUBR {} // No parameter passing for Subroutine
    ...
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=00;}
    Vector {         sig2=11;}
    Call MY_SUBR2 {} // No parameter passing for Subroutine
    WaveformTable MY_WFT;
    Vector {sig1=11;}
}
```

Pattern Compiler expands procedure PROC_STIL to be a straight line vectors and map procedure MY_SUBR and MY_SUBR2 to be test system subroutines. The corresponding Call vector is handled properly as well. The last vector is mapped to opcode RET in the procedure MY_SUBR. The Loop vectors of MY_SUBR2 are mapped to opcode LDC 94 and BNZ. The pseudo code is similar to:

```
Subroutine Memory:
    "MY_SUBR";
    INCMY_WFT {sig1=00; sig2=11;}
    RET MY_SUBR        MY_WFT {sig1=11; sig2=11;}
    "MY_SUBR2";
    SLC A              MY_WFT {sig1=00; sig2=11;} // SLC attached to
    previous vector
    INC                MY_WFT {sig1=10; sig2=10;} // Expand 1st Loop
    LDC 94             MY_WFT {sig1=00; sig2=01;}
    INC                MY_WFT {sig1=11; sig2=01;}
    MY_LOOP;
    INC                MY_WFT {sig1=10; sig2=10;}
    INC                MY_WFT {sig1=00; sig2=01;}
    BNZ MY_LOOP        MY_WFT {sig1=11; sig2=01;}
Main Memory:
    INC                MY_WFT {sig1=00; sig2=11;}
    INC                MY_WFT {sig1=00; sig2=11;} // Call attached on
    previous vector
    INC                MY_WFT {sig1=11; sig2=11;} // PROC_STIL #1
    INC                MY_WFT {sig1=00; sig2=11;} // PROC_STIL #2
    ...
    INC                MY_WFT {sig1=00; sig2=00;}
    CALL MY_SUBR       MY_WFT {sig1=01; sig2=10;}
    ...
    INC                MY_WFT {sig1=00; sig2=00;}
    CALL MY_SUBR2      MY_WFT {sig1=00; sig2=11;}
    INC                MY_WFT {sig1=11; sig2=11;}
```

2.2 Sync and Trigger Support

Pattern files do not need to know the complete definition of the triggers, they only need to know about the trigger names and types of the trigger (generate or waitfor). A pattern file contains (or includes) a STIL trigger definition that simply lists the trigger names along with their optional exit conditions. The latency pipe is also required when sync/triggers across different instruments are used. It provides enough latency to allow instrument-to-instrument synchronization. A latency pipe definition must be supplied before any type of trigger is used. This latency pipe definition is applicable to all types of triggers and is the integer number of the pipeline. The minimum length of the pipeline for the NPTLatencyPipeDef is 50 + confirmed sync (see Table 2-1) / period value. For example, the current test program utilizing five instruments with a test period of 5 ns must have a minimum latency pipeline length of 149 (50 + 495/5).

Table 2-1: Maximum Total Latency

Instruments	Confirmed Sync (ns)
5	495
10	645
15	795
20	945
25	1095
30	1245
35	1395
40	1545
50	1845
60	2145
70	2445
80	2745

In multiple time domain applications, different timing domains can be exited on the wait loop by matching with the ratio by using the exit condition of onRatio. For example, in a case where there are two time domains, and the patterns Pat A and Pat B have a 2:3 ratio between these two domains; the user may desire to exit on the boundary where the ratios match. Pat A always exits on an even number of cycles and Pat B can only exit on every third cycle to ensure these two domains are keeping the 2 to 3 ratio.

The user can also control the exit condition to only exit on a Dial a Pipe (DAP) boundary with the keyword onDAP. For example, if the DAP is 250, the match always exits on multiples of 250 cycles. If match happens on cycle 600, the match loop exits on cycle 750. These two exit conditions can be used separately or combined together.

The STIL extensions are defined as:

```
UserKeywords NPTTriggerDef NPTGenerateTrigger NPTWaitTrigger
NPTLatencyPipeDef;
NPTTriggerDef {(trigger_name (Exit = (onDAP | onRatio));)*}
NPTLatencyPipeDef {pipeline_length;}
```

Example 1: Possible NPTTriggerDef Syntax

```
NPTTriggerDef {aTrigger Exit=onDAP,onRatio;}
NPTTriggerDef {2Trigger Exit=onRatio;}
NPTTriggerDef {2Trigger Exit=onRatio; bTrigger;}
NPTTriggerDef {2Trigger Exit=onRatio; bTrigger Exit=onDAP;}
NPTTriggerDef {aTrigger; bTrigger; cTrigger;}
NPTTriggerDef {aTrigger;}
```

Example 2:

```
UserKeywords NPTTriggerDef NPTGenerateTrigger NPTWaitTrigger
NPTLatencyPipeDef;
NPTTriggerDef {"startMyTest"; "doMyTest"; "testIsDone";}
NPTTriggerDef {"endTest" Exit = onDAP,onRatio;}
NPTLatencyPipeDef {400;}
…
Pattern "TRY" {
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=11;}
    NPTWaitTrigger {"startMyTest";}
    Vector {sig1=00;}
    MatchLoop Infinite {
    Vector {sig1=01;}
    BreakPoint {}
    }
    Vector {sig1=11;}
    Vector {sig1=10;} // Generator trigger “DoMyTool” here
    NPTGenerateTrigger {"doMyTest";}
    Vector {sig1=11;}
    Vector {sig1=00;}
    Vector {sig1=01;} // Generate trigger “ToolIsDone” here
    NPTGenerateTrigger {"testIsDone";}
    Vector {sig1=11;}
    NPTWaitTrigger {"endTest";}
    Vector {sig1=00;}
    MatchLoop Infinite {
        Vector {sig1=01;}
        BreakPoint {Vector {}}
    }
    Vector {sig1=00;}
}
```

2.3 Continuous Call Application

The test system supports continuous as one flavor of sync/trigger support. Continuous Loop is mapped to WUS (Wait Until Sync) and continuous subroutine is mapped to CUS (Call Until Sync).

Example: Continuous Loop

```
UserKeywords NPTTriggerDef NPTWaitTrigger NPTLatencyPipeDef;
NPTTriggerDef {"CPUResumePattern";};
NPTLatencyPipeDef {400;}
...
Pattern "TRY" {
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=11;}
    NPTWaitTrigger {"CPUResumePattern";}
    Vector {sig1=00;}
    MatchLoop Infinite {
        Vector {sig1=01;}
        BreakPoint {Vector {}}
    }
    Vector {sig1=11;}
    Vector {sig1=00;}
}
```

The trigger CPUResumePattern consumes one SST (Select Sync Type) location. This trigger comes from the CPU in the test template code.

```
Waitfor Trigger Assignment (SST):
0 {"CPUResumePattern",0, 0}
```

Example: Continuous Subroutine

Any Call instruction proceeded by NPTWaitTrigger is translated to CUS, otherwise it is translated to a regular Call (if it is Subroutine Call) or is expanded to straight line (if it is STIL Procedure Call).

```
UserKeywords NPTSubroutine NPTTriggerDef NPTLatencyPipeDef NPTWaitTrigger;
NPTTriggerDef {"CPUResumePattern";}
NPTLatencyPipeDef {400;}

Procedure {
    NPTSubroutine "subrX";
    subrX {
        W Seq1;
        V {sig1=00; sig2=11;}
        V {sig1=10; sig2=01;}
        V {}
    }
}

Pattern "CUS_test" {
    W Seq1;
    V {sig1=00; sig2=11;}
    V {sig1=11;}
    V {sig1=01; sig2=10;}
    V {sig1=11;}
    NPTWaitTrigger {"CPUResumePattern";}
    V {sig1=00;} // CUS Vector
    Call subrX;
    V {sig1=00;}
    V {sig1=11;}
```

```
    V {sig1=00;}
    V {}
}
```

2.4 Match Vector

To support a feature similar to the RPTM <count> (Repeat the current vector until match condition is found or until a number of <count> is reached), XTOS uses the STIL statement MatchLoop and an additional extension to specify the Match sequence and pipeline length. The XTOS equivalent for the Repeat Until Match statement is called Wait Until Sync (WUS). The STIL equivalent statement for the Repeat statement is shown below:

```
MatchLoop <count | Infinite> {
    Vector { }
    BreakPoint {Vector { }}
}
```

STIL syntax requires the BreakPoint statement as the exit condition for a match vector. In XTOS, the BreakPoint statement is a do-nothing statement and the compiler does not generate any content from it. The BreakPoint statement is handled as a Null statement. Its presence in the MatchLoop statement is required to comply with the STIL standard only.

The STIL syntax allows Vector statement(s) inside the BreakPoint statement, but Sapphire NP's digital instruments do not support any extra vectors in the Match equivalent functionality. The presence of any Vector statement(s) is treated as an error in XTOS. The XTOS compiler's syntax extension to STIL MatchLoop and an example are shown below:

```
UserKeywords NPTMatchDef;
NPTMatchDef {Sequence = "<pass_fail_sequence>", Signal = "<signal_name>"+;}
```

Where:

- <*pass_fail_sequence*> is up to 32 P (pass) or F (fail) character string. For example, PFPFP means 5 match sequence with toggle Pass and Fail.
- <*signal_name*> is the signal reference to match on.

Example: Match Loop

```
UserKeywords NPTTriggerDef NPTMatchDef NPTLatencyPipeDef NPTWaitTrigger;
NPTTriggerDef {"MatchFound";}
NPTMatchDef {Sequence = "P", Signal_name = "Name1","Name2";}
NPTLatencyPipeDef {550;}
...
Pattern "TRY" {
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=11;}
    NPTWaitTrigger {"MatchFound";}
    Vector {sig1=00;}
    MatchLoop 3000 {
        Vector {sig1=01;}
        BreakPoint {}
```

```
    }
    Vector {sig1=11;}
    Vector {sig1=00;}
}
```

In addition to the vector statement usage example shown above, XTOS software also supports a Call statement in a NPTSubroutine. This feature is called a Call Until Sync (CUS). Any Call instruction preceded by a NPTWaitTrigger is translated to a CUS. A Call statement in the subroutine that does not meet this criterion (of being preceded by a NPTWaitTrigger) is translated to a regular Call or, is expanded into in-line vectors if it is STIL Procedure Call.

Example: Match Call

```
UserKeywords NPTTriggerDef NPTWaitTrigger NPTSubroutine NPTOpcode
NPTLatencyPipe NPTMatchDef;
NPTTriggerDef {"MatchFound";}
NPTMatchDef {Sequence="PFP", Signal_name = "Name3"};
NPTLatencyPipeDef {400;}

Procedure {
    NPTSubroutine "normal_subr";
    normal_subr {
        W Seq1;
            V {}
            V {}
            V {}
        }
    }
Pattern "CUS_test" {
    W Seq1;
        V {}
        V {}
        V {}
        V {}
        NPTWaitTrigger {"matchfound";}
        V {} // CUS vector
        Call normal_subr;
        V {}
        V {}
        V {}
        V {}
```

2.5 Keep Alive Vector

XTOS software supports the KeepAlive vector statement to allow the continuation of clock signals to keep the Device Under Test (DUT) in an pre-defined active state. This functionality is supported by a STIL extension syntax that identifies the KeepAlive vector in the specific pattern. Note that only one Vector statement is allowed to be a KeepAlive vector.

Syntax Extension:

```
UserKeywords NPTKeepAliveVector;
```

2.6 Parameter Passing

Parameter passing concept facilitates a mechanism to select functional data for a Signal either from the Subroutine or from the Main memory. This mechanism is supported as a part of the XTOS extension to the STIL standard. Parameter passing feature is supported for a per vector basis.

Syntax Extension:

```
UserKeywords NPTIncMain;
```

The NPTIncMain statement can be used in the subroutine to increment the main memory address that *converts* a regular Vector instruction into a parameter. The NPTIncMain results in the increment of main address. XTOS then retrieves the corresponding data from the main memory based on the user-defined symbol for this alternate source.

The main data source to be used is defined using a UserKeyword (NPTAlternativeDataSourceDef) statement as follows. Below is a complete example pattern showing the usage of the parameter passing concept:

```
UserKeywords NPTAlternativeDataSourceDef NPTIncMain NPTSubroutine;
NPTAlternativeDataSourceDef {Main = #;};
...
Procedures {
    PROC_STIL {
        WaveformTable MY_WFT;
        Vector {sig1=11; sig2=11;} // 2
        Vector {sig1=00;}           // 3
    }
    NPTSubroutine {MY_SUBR;}
    MY_SUBR {
        WaveformTable MY_WFT;
        NPTIncMain;                  // This will set "IncMain" in HW
        opcode
        Vector {sig1=11; sig2=#;}    // 6 - using sig2 =11;
        NPTIncMain;                  // This will set "IncMain" in HW
        opcode
        Vector {sig1=00; sig2=#;}    // 7 - using sig2=33;
        Vector {sig1=11;}            // 8
    }
Pattern TRY {
    WaveformTable MY_WFT;
    Vector {sig1=00; sig2=11;}   // 0
    Vector {sig1=00; sig2=11;}   // 1
    Call PROC_STIL {}
    Vector {sig1=00; sig2=11;}   // 4
    Vector {sig1=00; sig2=11;}   // 5
    Call MY_SUBR{};
    Vector {sig1=00; sig2=11;}   // - Param
    Vector {sig1=00; sig2= 33;}   // - Param
    Vector {sig1=00; sig2=11;}   // 9 - returns here after MY_SUBR
    Vector {sig1=00; sig2=11;}   // 10
    ...
}
```

In the above example, the pattern uses # to indicate sig2 Fdata is coming from Main data.

2.7 DC Channel Pattern Application

The 96-pin DC Channel instrument is one of the Sapphire NP instruments. This instrument provides a very limited timing pattern capability to the 96 channels on each instrument. The characteristics of the DC Channel instrument can be summarized as following:

- Four fixed period values (5 µs, 10 µs, 20 µs, 40 µs), no period switching on-the-fly.
- Ten fixed timing sequence per channel, (timing sequence can be changed on-the-fly):
  - NOP
  - Drive-Low at beginning of cycle
  - Drive-High at beginning of cycle
  - Drive-Z at beginning of cycle
  - Strobe-Low at end of cycle while Terminate-Low at beginning of cycle
  - Strobe-Low at end of cycle while Terminate-High at beginning of cycle
  - Strobe-Low at end of cycle while Terminate-Z at beginning of cycle
  - Strobe-High at end of cycle while Terminate-Low at beginning of cycle
  - Strobe-High at end of cycle while Terminate-High at beginning of cycle
  - Strobe-High at end of cycle while Terminate-Z at beginning of cycle
- There are no timing value control on these timing events.
- 8K vectors.
- Vector's instruction set supports INC, RPT, Trigger, and Wait for Sync.

2.7.1 Usage Model

The usage model considers both XTOS architecture impact and usage consistent between high-speed digital instruments and the DC Channel Instrument.

Even though there is no flexibility to program timing period values and timing edge values, a timing block is provided to satisfy system architecture. XTOS provides most of the timing block content. Every signal in the DC Channel instrument has these 10 timing sequences built-in, regardless of whether these timing sequences are used or not. The usage model of the timing block is:

- Pre-defined period name to select one of four pre-defined period values
- Every signal will have all 10 pre-defined timing sequences (always available).

The following are the pre-defined names for the four period values:

- Perod_5us
- Period_10us
- Period_20us
- Period_40us

The following are the pre-defined names for the ten timing sequences:

DCC_Nop	NOP.
DCC_D1	Drive-High at beginning of cycle.
DCC_D0	Drive-Low at beginning of cycle.
DCC_DZ	Drive-Z at beginning of cycle.
DCC_T1H	Strobe-High at end of cycle while Terminate-High at beginning of cycle.
DCC_T1L	Strobe-High at end of cycle while Terminate-Low at beginning of cycle.
DCC_T1Z	Strobe-High at end of cycle while Terminate-Z at beginning of cycle.
DCC_T0H	Strobe-Low at end of cycle while Terminate-High at beginning of cycle.
DCC_T0L	Strobe-Low at end of cycle while Terminate-Low at beginning of cycle.
DCC_T0Z	Strobe-Low at end of cycle while Terminate-Z at beginning of cycle.

An example of the Timing Block for the DCC pattern is shown below (only shows user programming section):

```
<Timing name = "norm_timing">
    <SignalGroupsRef name="sig_groups1" />
        <TimingDomain name="DCChannelDomain">
    <Period name="Period_1us">
        <Value1000e-9</Value>
    </Period>
    <SignalRef name="SIG_1"> </SignalRef>
        <SignalRef name="SIG_2"> </SignalRef>
    </TimingDomain>
</Timing>
```

The DC Channel pattern acts as one of multi-timing-domain patterns. The WaveformTable uses these pre-defined period, and sequence names with user defined WFC to construct the mapping. The pattern section uses the same source syntax defined in this document. The user has the flexibility to use any WFC for these pre-defined timing sequences. A Waveform Table block example is shown below:

```
Timing {
    WaveformTable MY_WFT {
        <Period_1us> Period '1000ns';
        Waveforms {
            SIG_1 {<DCC_T1L> L {'0ns' D; '995ns' h;}
                   <DCC_D0> 0 {'0ns' D;}
                      }
            SIG_2 {<DCC_DZ> a {'0ns' Z;}
                   <DCC_D0> b {'0ns' 0;}
```

```
                    <DCC_D1> c {'0ns' 1;}
                    <DCC_T0L} d {'0ns' D; '995ns' l;}
                    <DCC_T1L} e {'0ns' D; '995ns' h;}
                    <DCC_T0Z} f {'0ns' Z; '995ns' l;}
                    <DCC_T1Z} g {'0ns' Z; '995ns' h;}
                    }
        }
    }
}
```

A DC Channel pattern block example is shown below:

```
Pattern "TRY" {
    WaveformTable MY_WFT;
    Vector {SIG_1=L; SIG_2=g;}
    Vector {SIG_1=0; SIG_2=f;}
    Vector {SIG_1=L; SIG_2=a;}
    Vector {SIG_1=0; SIG_2=g;}
}

DCC Sequence Memory:
    ADV 4    MY_WFT

    DCC Pattern Data Memory
        {SIG_1=L; SIG_2=g;}
        {SIG_1=0; SIG_2=f;}
        {SIG_1=L; SIG_2=a;}
        {SIG_1=0; SIG_2=g;}
```

Scan Support 3

STIL has adopted the use of scan test patterns by various ATPG tools in many EDA environments. Scan pattern is a repeat process of scan load operation, followed by a launch and capture operation, and ends with a scan unload. The effect of scan in a scan pattern is often the result of the merge of the previous unload operation with the next load operation. The resulting scan pattern is typically shown as a scan load_unload operation, followed by a launch_capture operation. The load_unload operation is commonly implemented as a STIL Procedure or Macro.

3.1 ScanIn and ScanOut Declaration

A Signal or SignalGroups declaration supports scan operation with two attributes, ScanIn and ScanOut. The signal must be assigned one of these attributes to be allowed to operate as a scan signal.

```
SignalGroups {
        ( scan_in_name = 'signal_name' { ScanIn; } )*
        ( scan_out_name = 'signal_name' { ScanOut; } )*
}
```

3.2 Shift Statement

The main statement to support scan in STIL is the Shift statement, which contains one or more vectors required to shift one scan event in or out of scan chains. Vectors within the Shift block use a special waveform character (#) in signal assignments to indicate that scan data is to be substituted in the vector for the signal. The values to be substituted are defined in the Call or Macro invocation.

The scan data passed in the Shift block is presented in left-to-right order; the left-most waveform character is the first data for that signal.

```
Procedures {
    "load_unload" {
        W "scan_timing";
        C { SI = 0000000000; SO = XXXXXXXXXX; }
        Shift {
```

```
            V { SI = ##########; SO = ##########; scan_clk = 1; }
        }
    }
}

Pattern "scan_pat" {
    W "scan_timing";
    V { all_pins = 0010XXXX0011100HHLHXXX01010; }
    Call "load_unload" { SI = 0101010101; }
    V { in_pins = 1011101110010110; out_pins = HHLLLLHH; }
    Call "load_unload" { SI = 1110001110; SO = HHHLLLHHHL; }
    V { in_pins = 1000000111010000; out_pins = HLXXXXLH; }
    Call "load_unload" { SO = HLLLLHLLLL; }
    }
}
```

3.3 ScanStructure Block

The optional ScanStructure block defines the scan chain content to support scan testing. The details have been described in “ScanStructures Block” on page 1-6

3.4 Scan Pattern Examples

There are several styles of programing a scan pattern. Each style has its own strengths and flexibility. The following are few examples.

3.4.1 Example 1

Scan Pattern with MacroDefs, but without ScanStructures block:

```
STIL 1.0 { TRC D08; }

Header {
   Title " Simple Scan Pattern #1 - No ScanStructure ";
   Date "Sun  Dec 13 22:10:00 2003";
}

Signals {
   sig1 In;   sig2 In;   bus1 InOut;  bus2 InOut;
   sig3 In;   sig4 In;   sig5 In;
   sig6 Out;  sig7 Out;  sig8 Out;
}

SignalGroups {
   si1 = 'sig3' { ScanIn 20; }        // Note 1)
   si2 = 'sig4' { ScanIn 30; }
   si3 = 'sig5' { ScanIn 25; }
   so1 = 'sig6' { ScanOut 20; }
   so2 = 'sig7' { ScanOut 30; }
```

```
   so3 = 'sig8' { ScanOut 25; }
   scan_in  = 'si1 + si2 + si3';      // Note 2)
   scan_out = 'so1 + so2 + so3';
   pa_pin   = 'sig1 + sig2 + bus1 + bus2';
   all_pin  = 'pa_pin + scan_in + scan_out';
}

Timing T1 {
   WaveformTable W1 {
      Period '100ns';
      Waveforms {
         scan_in  { <in_sc>  01   { '0ns' D/U ; }}   // Note 3)
         scan_out { <ou_sc>  LHX  { '0ns' L/H/X ; }}
         pa_pin   { <in_seq> 01   { '0ns' D/U ; }
                    <z_seq>   Z   { '0ns' Z ; }
                    <ou_seq> LH   { '0ns' L/H ; }}
      }
   } // end WaveformTable W1
} // end Timing T1

MacroDefs
{
   all_scans {
      W W1;
      // Note 4)
      C { si1=0; si2=0; si3=0; so1=X; so2=X; so3=X; pa_pin=####; }
      Shift { V { scan_in=###; scan_out=###; } }     // Note 5)
   }
}

PatternBurst burst1 {
   PatList { pat1; }
} // end PatternBurst "hello tester burst"

PatternExec {
   Timing T1;
   PatternBurst burst1;
} // end PatternExec

Pattern pat1 {
   W W1;
   V { all_pin = 00ZZ000XXX; }
   scan_pat_0:
   Macro all_scans { pa_pin=10ZZ;                    // Note 6)
      si1=10101111110000011001;
      si2=11111000001111100000111110000;
      si3=11111111111111111111111;
   }
   V { all_pin = 01HH100HHL; }
   V { all_pin = 11LH011LXH; }
   scan_pat_1:
   Macro all_scans { pa_pin=10ZZ;
      si1=10101111110000011001;
```

```
      si2=1111100000111111000001111100000;
      si3=111111111111111111111111;
      so1=HLHHHLXXLLXXXXLLLLHH;
      so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
      so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
   V { all_pin = 10HH111HHL; }
   V { all_pin = 01ZZ001LHX; }
   V { all_pin = 0011000XXX; }
   scan_pat_2:
   Macro all_scans { pa_pin=10LL;
      si1=10101111110000011001;
      si2=1111100000111111000001111100000;
      si3=111111111111111111111111;
      so1=HLHHHLXXLLXXXXLLLLHH;
      so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
      so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
   V { all_pin = 0000111LLL; }
   scan_pat_3:
   Macro all_scans { pa_pin=10ZZ;
       so1=HLHHHLXXLLXXXXLLLLHH;
       so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
       so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
} // end pattern "pat1"
```

Notes:

1. SignalGroups: Some of signals are declared as scan input port with keyword "ScanIn" and some signals are declared as scan output port with keyword "ScanOut". The declaration can be part of Signals block or SignalGroups block. The second parameter defines the length of each scan chain associated with the scan port. It should have matched length for the corresponding scan inputs and outputs (for example, si1 and so1).
2. SignalGroup (scan_in, scan_out): Scan shift chain has its own input and output port and its data is always standalone. It may have scan input and output group for the convenience of defining a scan Procedure or Macro. The signal group can also be used in WaveformTable, but not in the data assignment.
3. WaveformTable: The scan input signals are always using WFC 0 to represent drive-low state and 1 to represent drive-high state. It should be combined as 01 event list to imply F1 mode. The scan output signals should use WFC LHX to imply FM mode while L means strobe-low, H means strobe-high, and X means do-not-care. The non-scan signals just use 01 for drive cycle and LH for strobe cycle.
4. Condition statement: The scan operation can be defined as macro or procedure. In the Macro case, it should have a Condition (C) statement to define the default state of each signals. The default states will provide scan signals' padding WFC and broadside signals' WFC. Broadside signals (pa_pin) use the # symbol to allow the passing of WFC per macro call.

5. Shift statement: The Shift statement defines the scan shift operation. The ### means to apply # to each one of the signals in the signal group. The # will consume scan shift data per scan length of each scan chain. The compiler will pad each scan chain to the maximum length.
6. Macro statement: The Macro statement invokes the scan operation by expanding the macro "all_scans". Notice the very first scan operation (scan_pat_0:) only has scan input chain, scan output chain is filled with "X" (by the compiler). The very last scan operation (scan_pat_3:) only has scan output chain, scan inputs are filled with "0" (by the compiler). The Macro statement also provides the static state of each broadside signal (pa_pin).

3.4.2 Example 2

Scan pattern with MacroDefs and ScanStructures blocks:

```
STIL 1.0 { TRC D08; }

Header {
   Title " Simple Scan Pattern #2 - with ScanStructure ";
   Date "Sun  Dec 13 22:00:00 2003";
}

Signals {
   sig1 In;   sig2 In;   bus1 InOut;  bus2 InOut;
   sig3 In;   sig4 In;   sig5 In;
   sig6 Out;  sig7 Out;  sig8 Out;
}

SignalGroups {
   si1 = 'sig3' { ScanIn; }           // Note 1)
   si2 = 'sig4' { ScanIn; }
   si3 = 'sig5' { ScanIn; }
   so1 = 'sig6' { ScanOut; }
   so2 = 'sig7' { ScanOut; }
   so3 = 'sig8' { ScanOut; }
   scan_in  = 'si1 + si2 + si3';
   scan_out = 'so1 + so2 + so3';
   pa_pin   = 'sig1 + sig2 + bus1 + bus2';
   all_pin  = 'pa_pin + scan_in + scan_out';
}

Timing T1 {
   WaveformTable W1 {
      Period '100ns';
      Waveforms {
         scan_in  { <in_sc>  01   { '0ns' D/U ; }}
         scan_out { <ou_sc>  LHX  { '0ns' L/H/X ; }}
         pa_pin   { <in_seq> 01   { '0ns' D/U ; }
                    <z_seq>   Z   { '0ns' Z ; }
                    <ou_seq> LH   { '0ns' L/H ; }}
      }
```

```
   } // end WaveformTable W1
} // end Timing T1

ScanStructures {                        // Note 2)
   ScanChain "SCH.1.33" {
      ScanLength 20;
      ScanIn "si1";
      ScanOut "so1";
      ScanInversion 0;
      ScanCells "CH.1.1" "CH.1.2" "CH.1.3" "CH.1.4"
      "CH.1.5" "CH.1.6" "CH.1.7" "CH.1.8" "CH.1.9"
      "CH.1.10" "CH.1.11" "CH.1.12" "CH.1.13"
      "CH.1.14" "CH.1.15" "CH.1.16" "CH.1.17"
      "CH.1.18" "CH.1.19" "CH.1.20" ;
   }
   ScanChain "SCH.1.37" {
      ScanLength 30;
      ScanIn "si2";
      ScanOut "so2";
      ScanInversion 0;
      ScanCells "CH.2.1" "CH.2.2" "CH.2.3" "CH.2.4"
      "CH.2.5" "CH.2.6" "CH.2.7" "CH.2.8" "CH.2.9"
      "CH.2.10" "CH.2.11" "CH.2.12" "CH.2.13"
      "CH.2.14" "CH.2.15" "CH.2.16" "CH.2.17"
      "CH.2.18" "CH.2.19" "CH.2.20" "CH.2.21"
      "CH.2.22" "CH.2.23" "CH.2.24" "CH.2.25"
      "CH.2.26" "CH.2.27" "CH.2.28" "CH.2.29"
      "CH.2.30" ;
   }
   ScanChain "SCH.1.16" {
      ScanLength 25;
      ScanIn "si3";
      ScanOut "so3";
      ScanInversion 0;
      ScanCells "CH.3.1" "CH.3.2" "CH.3.3" "CH.3.4"
      "CH.3.5" "CH.3.6" "CH.3.7" "CH.3.8" "CH.3.9"
      "CH.3.10" "CH.3.11" "CH.3.12" "CH.3.13"
      "CH.3.14" "CH.3.15" "CH.3.16" "CH.3.17"
      "CH.3.18" "CH.3.19" "CH.3.20" "CH.3.21"
      "CH.3.22" "CH.3.23" "CH.3.24" "CH.3.25" ;
   }
}

MacroDefs
{
   all_scans {
      W W1;                             // Note 3)
      C { si1=0; si2=0; si3=0; so1=X; so2=X; so3=X; pa_pin=10ZZ; }
      Shift { V { scan_in=\r3 #; scan_out=\r3 #; } }// Note 4)
   }
}

PatternBurst burst1 {
```

```
   PatList { pat2; }
} // end PatternBurst "hello tester burst"

PatternExec {
   Timing T1;
   PatternBurst burst1;
} // end PatternExec

Pattern pat2 {
   W W1;
   V { all_pin = 00ZZ000XXX; }
   scan_pat_0:
   Macro all_scans {
      si1=10101111110000011001;
      si2=11111000001111100000111100000;
      si3=11111111111111111111111111;
   }
   V { all_pin = 01HH100HHL; }
   V { all_pin = 11LH011LXH; }
   scan_pat_1:
   Macro all_scans {
      si1=10101111110000011001;
      si2=11111000001111100000111100000;
      si3=11111111111111111111111111;
      so1=HLHHHLXXLLXXXXLLLLHH;
      so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
      so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
   V { all_pin = 10HH111HHL; }
   V { all_pin = 01ZZ001LHX; }
   V { all_pin = 0011000XXX; }
   scan_pat_2:
   Macro all_scans {
      si1=10101111110000011001;
      si2=11111000001111100000111100000;
      si3=11111111111111111111111111;
      so1=HLHHHLXXLLXXXXLLLLHH;
      so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
      so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
   V { all_pin = 0000111LLL; }
   scan_pat_3:
   Macro all_scans {
       so1=HLHHHLXXLLXXXXLLLLHH;
       so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
       so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
} // end pattern "pat2"
```

Notes:

1. SignalGroups: Some of signals are declared as scan input port with keyword "ScanIn" and some signals are declared as scan output port with keyword "ScanOut". The declaration can be part of Signals block or SignalGroups block. No ScanLength is declared here because it is declared in the ScanStructure.
2. ScanStructures: This structure holds the definition of scan shift chain. It defines the length of each scan shift chain ("ScanLength"). Based on this information the compiler will pad each scan chain to make all scan shift chains the maximum length (in this example, it is 30). The padding is based on the Condition statement in the macro. The padding will be provided to the beginning of each input scan chain and to the ending of each output scan chain.
3. Condition statement: The scan operation is defined as macro. In the Condition (C) statement, the default states provide scan signals padding WFC and broadside signals WFC. (No passing in broadside states in this example, if broadside is always fixed states.)
4. Shift statement: The Shift statement defines the scan shift operation.The \r# means repeat the following WFC # times. This short hand notation provides the convenience of applying the shift instruction to all three scan inputs and outputs. In other words \r3 # == ###. (If there are 100 scan shift chains, then we can use \r100 #.)

3.4.3 Example 3

Scan pattern with Procedure, but without ScanStructures block:

```
STIL 1.0 { TRC D08; }

Header {
   Title " Simple Scan Pattern #3 - No ScanStructure/Use Procedure ";
   Date "Sun  Dec 14 14:10:00 2003";
}

Signals {
   sig1 In;   sig2 In;   bus1 InOut;  bus2 InOut;
   sig3 In;   sig4 In;   sig5 In;
   sig6 Out;  sig7 Out;  sig8 Out;
}

SignalGroups {
   si1 = 'sig3' { ScanIn 20; }
   si2 = 'sig4' { ScanIn 30; }
   si3 = 'sig5' { ScanIn 25; }
   so1 = 'sig6' { ScanOut 20; }
   so2 = 'sig7' { ScanOut 30; }
   so3 = 'sig8' { ScanOut 25; }
   scan_in  = 'si1 + si2 + si3';
   scan_out = 'so1 + so2 + so3';
   pa_pin   = 'sig1 + sig2 + bus1 + bus2';
   all_pin  = 'pa_pin + scan_in + scan_out';
```

```
}

Timing T1 {
   WaveformTable W1 {
      Period '100ns';
      Waveforms {
         scan_in  { <in_sc>  01   { '0ns' D/U ; }}
         scan_out { <ou_sc>  LHX  { '0ns' L/H/X ; }}
         pa_pin   { <in_seq> 01   { '0ns' D/U ; }
                    <z_seq>   Z   { '0ns' Z ; }
                    <ou_seq> LH   { '0ns' L/H ; }}
      }
   } // end WaveformTable W1
} // end Timing T1

Procedures
{
   all_scans {                          // Note 1)
      W W1;
      C { pa_pin=####; scan_in=000; scan_out=XXX; }
      Shift { V { scan_in=\r3 #; scan_out=\r3 #; } }
   }
}

PatternBurst burst3 {
   PatList { pat3; }
} // end PatternBurst "hello tester burst"

PatternExec {
   Timing T1;
   PatternBurst burst3;
} // end PatternExec

Pattern pat3 {
   W W1;
   V { all_pin = 00ZZ000XXX; }
   scan_pat_0:
   Call all_scans { pa_pin=10ZZ;      // Note 2)
      si1=10101111110000011001;
      si2=11111000001111100000111110000;
      si3=11111111111111111111111111;
   }
   V { all_pin = 01HH100HHL; }
   V { all_pin = 11LH011LXH; }
   scan_pat_1:
   Call all_scans { pa_pin=00LL;
      si1=10101111110000011001;
      si2=11111000001111100000111110000;
      si3=11111111111111111111111111;
      so1=HLHHHLXXLLXXXXLLLLHH;
      so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
      so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
   }
```

```
    V { all_pin = 10HH111HHL; }
    V { all_pin = 01ZZ001LHX; }
    V { all_pin = 0011000XXX; }
    scan_pat_2:
    Call all_scans { pa_pin=00HH;
       si1=10101111110000011001;
       si2=11111000001111100000111110000 0;
       si3=1111111111111111111111111;
       so1=HLHHHLXXLLXXXXLLLLHH;
       so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
       so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
    }
    V { all_pin = 0000111LLL; }
    scan_pat_3:
    Call all_scans { pa_pin=00ZZ;
        so1=HLHHHLXXLLXXXXLLLLHH;
        so2=HHHHLLLLLLXXXXXXXXLLHHLLHHHXXL;
        so3=LLLXXXXHHHXXXLHHHLLLHHHXL;
    }
} // end pattern "pat3"
```

Notes:

1. Procedures: In this Procedures case, it should have a Condition (C) statement to define the default state of each signal.The default states will provide scan signals' padding WFC and broadside signals' WFC.
2. Call statement: The Call statement invokes scan operations by expanding the procedure "all_scans". Notice the very first scan operation (scan_pat_0:) only has scan input chain, scan output chain is filled with "X" (by the Compiler). The very last scan operation (scan_pat_3:) only has scan output chain, scan inputs will be filled with "0" (by the Compiler). The signal WFC will be back to prior call statement for a non-defined signal for a procedure.

Analog Support 4

Analog support in the Sapphire NP system is provided through alternative data source selection process. A set of different symbols may be defined as STIL extension syntax to support various data sources. These symbols are used to distinguish the alternative data source between analog source and analog capture. It is important to note that within a pin slice, all analog source signals source at the same time and all analog capture signals capture at the same time. However, the analog source signals and capture signals do not have to be used at the same time. This results in three different alternative data selection combinations involving analog signals within one pin slice. They are all source, all capture, and all source and capture.

The XTOS STIL complier support for the analog application is provided using the waveform characters in the WaveformTable. This is accomplished by defining dedicated timing sequences when alternative data source selection is needed in the pattern. Two new keywords, NPTAnalogSource and NPTAnalogCapture are used to define digitized analog applications. These keywords are used in front of the event list to mark the current event list and WFC to be used for analog application. Only one waveform character is allowed for the event list since the analog instrument and not the digital pattern memory provides the pattern data.

4.1 Digitized Analog Pattern Example

The user can mark any sequence containing a single waveform character with the use of NPTAnalogSource or NPTAnalogCapture to define the corresponding signal to get data from an analog instrument. Note that a given sequence can only be marked as either source or capture and not both.

The following is an example of the analog pattern:

```
UserKeywords NPTAnalogSource NPTAnalogCapture;

Timing "MY_TIMING" {
   WaveformTable one {
        <PERIOD_1> Period '500ns';
        Waveforms {
           sig1 { <SEQ_Dr> A   { NPTAnalogSource; '0ns' D/U; } }
```

```
            sig1 { <SEQ_Dr> 01 { '0ns' D/U; } }
            sig2 { <SEQ_Cmp> C   { NPTAnalogCapture; '0ns' L/H; } }
            sig2 { <SEQ_Cmp> LH  { '0ns' L/H; } }
            sig3 { <SEQ_Dr1> K   { NPTAnalogSource; '10ns' D/U; } }
            sig3 { <SEQ_Dr2> 01 { '50ns' D/U; } }
            sig4 { <SEQ_Cmp1> J   { NPTAnalogCapture; '10ns' L/H; } }
            sig4 { <SEQ_Cmp2> LH  { '80ns' L/H; } }
        }
    } // end WaveformTable one
}

...
Pattern "TRY" {
WaveformTable MY_WFT;
Vector { sig1=00; sig2=HH; sig3=11; sig4=HH; }
Vector { sig1=A; sig2=LH; sig3=K; }
Vector { sig1=11; sig3=10; }
Vector { sig1=A; sig2=C; sig3=K; sig4=J; }
Vector { sig1=00; sig2=LL; sig3=00; }
Vector { sig2=C; sig4=J; }
Vector { sig1=11; sig2=HL; sig4=LL; }
Vector { sig1=A; sig3=K; }
Vector { sig1=00; sig2=LL; sig3=00; }
Vector { sig2=C; sig4=J; }
Vector { sig2=HH; sig3=11; sig4=LL; }
}
```

This pattern uses waveform character A and K to indicate that the corresponding signal is sourcing from an analog source. C and J are used to indicate the corresponding signal is capturing for the current vector.

XTOS Pattern Compiler and Composite Linker 5

This chapter covers the pattern complier and the composite linker that accompany XTOS.

5.1 Pattern Complier

XTOS pattern compiler can be invoked from a Command Line Interface in a Command Prompt window.

The Command Line Interface for the pattern compiler is available for the Windows 2000® and Solaris® operating systems.

5.1.1 XTOS Pattern Complier Command Line Interface—Windows 2000

The command line interface for the XTOS pattern compiler is located at:

\XTOS*release_number*\\bin\CompileStil.exe

Compilation of a pattern in XTOS software results in a pattern object directory with the pattern object files to be used by various features that access the pattern data. An example structure of a pattern object directory is:

Source file PAT1.STIL

Upon compiling the above file a PAT1.pof directory is created with a vpr, rpr, lpi, and a text file containing the compiler statistics. The first three files are binary objects used by the XTOS software, while the fourth one is for the user's information.

Typing the compile command with the --help parameter displays the following command usage for the pattern compiler:

```
usage: \XTOS\release_number\\bin\CompileStil.exe [options] <STILFileName>
Required Arguments:
    <STILFileName> STIL file to compile
Optional Arguments:
    --diag-output  Print diagnostic messages
    --gen-image-only Regenerate pattern image only
    --help         Display this usage message
    --namemap <NameMapXMLFile> Name map XML input file
```

```
--nospin       Don't show spinning progress indicator on console
--parse-only   Parse the STIL but don't generate a pattern image
--showstats    Print compilations statistics after compiling
--verbose      Print lots of program messages
```

Without any of the optional arguments, the compile command displays a spinning progress indicator on the console. The optional arguments produce specific informational messages as indicated:

--diag-output — This option prints diagnostics messages such as the elapsed time for parsing phase and the pattern image generation.

--parse-only — This option only performs the parsing phase of the compilation resulting in the generation of the pattern object file directory with only virtual pattern representation (vpr) file and the statistics text file (CompileStats.txt).

--gen-image-only — This option re-generates the pattern image provided that the pattern object file (directory) already exists (previously parsed).

--namemap — This option uses an XML input file to be used to map the signal pin definitions as described in the test program. The file name here is the one that contains the name map XML block's name used by the test program.

--showstats — This option prints the compilation process's statistical information such as the Vector Type Select definitions and procedure call statistics.

--verbose — This option displays the compilation process details such as the signal names associated with each Event Sequence Store location.

--nospin — This option silently performs the compilation without the spinning progress indicator and the elapsed time information.

--nolink — This option overrides the default behavior of the complier so that a composite image is not generated (only a loadable pattern image is generated).

Please refer to the *XTOS Software Tools Manual*, (P/N 57510507) for details on name map XML block.

5.1.2 NameMap Support in XTOS Pattern Compiler

The XTOS pattern compiler uses an XML NameMap block that is associated with the test program to define blocks containing references of STIL names to names defined in the XTOS test program. This is a required argument to be used with the XTOS pattern compiler (CompileStil.exe). An example of the NameMap file used is:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Created by XML decompiler version 1.0 -->
<blocks>
    <NameMaps name="Loadboard_1">
        <Sites count="1">
            <ThreadModel>Single</ThreadModel>
```

```
            </Sites>
            <SignalRef name="Abus_0">
                <SlotPogo>7.0</SlotPogo>
            </SignalRef>
            <SignalRef name="Abus_1">
                <SlotPogo>7.4</SlotPogo>
            </SignalRef>
            <SignalRef name="Bbus_0">
                <SlotPogo>7.2</SlotPogo>
            </SignalRef>
            <SignalRef name="Bbus_1">
                <SlotPogo>7.6</SlotPogo>
            </SignalRef>
            <SignalRef name="Cbus_0">
                <SlotPogo>8.0</SlotPogo>
            </SignalRef>
            <SignalRef name="Cbus_1">
                <SlotPogo>8.4</SlotPogo>
            </SignalRef>
            <SignalRef name="Dbus_0">
                <SlotPogo>8.2</SlotPogo>
            </SignalRef>
            <SignalRef name="Dbus_1">
                <SlotPogo>8.6</SlotPogo>
            </SignalRef>
        </NameMaps>
    </blocks>
```

An example command file that compiles and then creates a composite image is:

```
CompileStil --namemap Loadboard_1.xml" .\vec_dir\Pass_all_F2.stil
CompileStil --namemap Loadboard_1.xml" .\vec_dir\Clock_both_F2.stil
CompositeLink .\vec_dir\Comp_F2.xml
```

5.1.3 XTOS Pattern Complier Command Line Interface—Solaris

The command line interface for the XTOS pattern compiler is located at:

/XTOS/*release_number*/bin/CompileStil.exe

Compilation of a pattern in XTOS software results in a pattern object directory with the pattern object files to be used by various features that access the pattern data. An example structure of a pattern object directory is:

Source file PAT1.STIL

Upon compiling the above file a PAT1.pof directory is created with a vpr, rpr, lpi, and a text file containing the compiler statistics. The first three files are binary objects used by the XTOS software, while the fourth one is for the user's information.

Typing the compile command with the --help parameter displays the following command usage for the pattern compiler:

```
usage: /XTOS/release_number/bin/CompileStil.exe [options] <STILFileName>
Required Arguments:
    <STILFileName> STIL file to compile
```

```
Optional Arguments:
    --diag-output  Print diagnostic messages
    --gen-image-only Regenerate pattern image only
    --help         Display this usage message
    --namemap <NameMapXMLFile> Name map XML input file
    --nospin       Don't show spinning progress indicator on console
    --parse-only   Parse the STIL but don't generate a pattern image
    --showstats    Print compilations statistics after compiling
    --verbose      Print lots of program messages
```

Without any of the optional arguments, the compile command displays a spinning progress indicator on the console. The optional arguments produce specific informational messages as indicated:

--diag-output — This option prints diagnostics messages such as the elapsed time for parsing phase and the pattern image generation.

--parse-only — This option only performs the parsing phase of the compilation resulting in the generation of the pattern object file directory with only virtual pattern representation (vpr) file and the statistics text file (CompileStats.txt).

--gen-image-only — This option re-generates the pattern image provided that the pattern object file (directory) already exists (previously parsed).

--namemap — This option uses an XML input file to be used to map the signal pin definitions as described in the test program. The file name here is the one that contains the name map XML block's name used by the test program.

--showstats — This option prints the compilation process's statistical information such as the Vector Type Select definitions and procedure call statistics.

--verbose — This option displays the compilation process details such as the signal names associated with each Event Sequence Store location.

--nospin — This option silently performs the compilation without the spinning progress indicator and the elapsed time information.

--nolink — This option overrides the default behavior of the complier so that a composite image is not generated (only a loadable pattern image is generated).

Please refer to the *XTOS Software Tools Manual*, (P/N 57510507) for details on name map XML block.

5.2 Pattern Decomplier

XTOS pattern decompiler can be invoked from a Command Line Interface in a Command Prompt window. XTOS pattern decompiler decompiles the vpr (virtual pattern representation) file into the source STIL file.

The pattern decompiler is available for the Windows 2000 and Solaris operating systems.

5.2.1 XTOS Pattern Decomplier Command Line Interface—Windows 2000

The command line interface for the XTOS pattern decompiler is located at:

\XTOS*release_number*\\bin\DecompiletoStil.exe

Decompilation of a pattern file in XTOS software results in a STIL source file of the same name.

Typing the decompile command with the --help parameter displays the following command usage for the pattern decompiler:

```
usage: \XTOS\release_number\bin\DecompiletoStil.exe [options] <POFFileName>
Required Arguments:
    <POFFileName> pattern object file to decompile
Optional Arguments:
    --diag-output  Print diagnostic messages
    --help         Display this usage message
    --verbose      Print lots of program messages
```

Without any of the optional arguments, the decompile command displays a spinning progress indicator on the console. The optional arguments produce specific informational messages as indicated:

--diag-output — This option prints diagnostics messages such as the elapsed time for parsing phase and the pattern image generation.

--verbose — This option displays the compilation process details such as the signal names associated with each Event Sequence Store location.

5.2.2 XTOS Pattern Decomplier Command Line Interface—Solaris

The command line interface for the XTOS pattern decompiler is located at:

/XTOS/*release_number*/bin/DecompiletoStil.exe

Decompilation of a pattern file in XTOS software results in a STIL source file of the same name.

Typing the decompile command with the --help parameter displays the following command usage for the pattern decompiler:

```
usage: \XTOS\release_number\bin\DecompiletoStil.exe [options] <POFFileName>
Required Arguments:
    <POFFileName> pattern object file to decompile
Optional Arguments:
    --diag-output  Print diagnostic messages
    --help         Display this usage message
    --verbose      Print lots of program messages
```

Without any of the optional arguments, the decompile command displays a spinning progress indicator on the console. The optional arguments produce specific informational messages as indicated:

--diag-output — This option prints diagnostics messages such as the elapsed time for parsing phase and the pattern image generation.

--verbose	This option displays the compilation process details such as the signal names associated with each Event Sequence Store location.

5.3 XTOS Pattern Composite Linker

XTOS software provides facilities to accommodate patterns that contain resources that are shared between multiple patterns. These patterns contain references to, and information for, a number of *shared resources* such as main Event Sequence Store Memory (ESSM), Source Data Select (SDS), Select Sync Type (SST), and SUBROUTINE_F. In order to be burstable (execute all patterns together as one pattern), every pattern in the burst has to be consistent in the references to the shared resources.

The CompositeLink tool takes one or more individual patterns and makes a *composite image* of the *sharable* resources for them, along with a set of *remap tables* so that the references into those resources can be adjusted in the patterns at the pattern load-time.

The composite image contains the composite image for each of the shared resources, a list of the patterns in that *composite* group, and the maps for each of the patterns for each resource.

The Composite block defines the patterns that may be chained together and the patterns that must always be run in parallel (as part of an multiple time domain pattern execution). The following rules are applicable to the composite:

1. A given pattern may be used in multiple Composite blocks.
2. Composite blocks define the patterns that may be chained together.
3. Composite blocks define pattern domains—the division of a pattern into multiple sections.
4. Composite blocks define the sub-patterns in each pattern sections.
5. PatternLists within PatternBurst blocks may list either pattern name or sub-pattern names of the primary section (the very first domain in the case of multiple time domain patterns).
6. PatternLists within PattenBurst blocks always refers to one and only one Composite block.
7. Identical PatternLists may be declared in multiple PatternBursts, each referring to a different Composite block.

The following example illustrates a multiple time domain group of patterns consisting of three patterns. Each pattern has three *sections*. Each section exists in a different *domain*. A domain is associated with unique tester resources—these resources are defined in the pattern files. The resources, in turn, determine the timing domain that the patterns are executed.

Below is a Composite block layout example:

```
Pattern     Domain1            Domain2             Domain3
            FullTime           HalfTime Section    QuarterTime Sections
            Section

Multiply    PatF1              PatM1               PatQ1

Divide      PatF2              PatM2               PatQ2

Boolean     PatF3              PatM3               PatQ3
```

In the above example, Multiply, Divide, and Boolean are three user names referring to the three sections of Pattern. Usage of user names is optional. Patterns PatF1, PatM1, and PatQ1 belong to the user name Multiply. Patterns PatF2, PatM2, and PatQ2 belong to the user name Divide. Patterns PatF3, PatM3, and PatQ3 belong to the user name Boolean. A composite block may only contain the Pattern sections. It is important to note that in a multiple time domain composite the number of patterns in each domain must be the same.

An example (XML) block syntax for the above composite block is shown below. This results in three composites named FullTime, HalfTime, and QuarterTime.

```
<Composite name="myComposite">
       <PatternList> // this list is optional; if specified, then these
                        //names may be used in the PatternBurst list
              <Pattern name="Multiply"><\Pattern >
              <Pattern name="Divide"><\Pattern >
              <Pattern name="Boolean"><\Pattern >
       </PatternList>
       <DomainList>
              <Domain name="FullTime">
                            <Stilfile>PatF1</Stilfile>
                            <Stilfile>PatF2</Stilfile>
                            <Stilfile>PatF3</Stilfile>
              </Domain>
              <Domain name="HalfTime">
                            <Stilfile>PatF1</Stilfile>
                            <Stilfile>PatF2</Stilfile>
                            <Stilfile>PatF3</Stilfile>
              </Domain>
              <Domain name="QuarterTime">
                            <Stilfile>PatF1</Stilfile>
                            <Stilfile>PatF2</Stilfile>
                            <Stilfile>PatF3</Stilfile>
              </Domain>
       </DomainList>
</Composite>
```

The creation of a composite image is achieved in two steps:

- First the patterns need to be compiled using the XTOS pattern compiler (CompileStil). By default the pattern compiler creates a composite image for the single pattern being complied. This is overridden with a --nolink option.

- Secondly, the application CompositeLink (\XTOS*release_number*\\bin\CompositeLink.exe) should be executed with a composite XML block as an input argument.

Please refer to the Command Line Interface (see “XTOS Pattern Composite Linker” on page 5-6) section for details on invoking the CompositeLink application (the compositer). The compositer processes all the patterns within a composite block and creates a composite image.

5.3.1 Compositing Patterns

An example composite file (Comp_F2.xml) is shown below. In this example, the composite linker creates a composite image containing the shared resource for patterns Pass_all_F2.stil and Clock_both_F2.stil file.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<blocks>
    <Composite name="Comp_F2">
        <DomainList>
            <Domain name="Primary">
                <Stilfile>Pass_all_F2</Stilfile>
                <Stilfile>Clock_both_F2</Stilfile>
            </Domain>
        </DomainList>
    </Composite>
</blocks>
```

For the example above, the CompositeLink application creates a directory named Comp_F2.cdir inside which there are two sub-directories named Pass_all_F2.cpof and Clock_both_F2.cpof. Each of these cpof (composite pattern object file) directories contain their respective composited LPI (Loadable Pattern Image) and RPR (Runtime Pattern Repository) files that are used subsequently by the XTOS pattern loader and other components. It is required that the two patterns be compiled before the CompositeLink application is executed on the composite file.

The above example illustrated the compositing of patterns that belong to a single time domain application. XTOS supports the compositing of the multiple time domain patterns through the same mechanism. The following example shows the composite block structure for an application with two timing domains named Primary (contains pattern files MyPat1_pri.stil and MyPat2_pri.stil) and Secondary (contains pattern files MyPat1_sec.stil and MyPat2_sec.stil). This concept is applicable to multiple time domain.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<blocks>
    <Composite name="Comp_F1">
        <DomainList>
            <Domain name="Primary">
                <Stilfile>MyPat1_pri</Stilfile>
                <Stilfile>MyPat2_pri</Stilfile>
            </Domain>
            <Domain name="secondary">
                <Stilfile>MyPat1_sec</Stilfile>
```

```
                <Stilfile>MyPat2_sec</Stilfile>
            </Domain>
        </DomainList>
    </Composite>
</blocks>
```

5.3.2 XTOS Pattern CompositeLink Command Line Interface—Windows 2000

The command line interface for the XTOS Pattern compositor is as follows:

\XTOS*release_number*\bin\CompositeLink.exe

Typing the above command with the --help parameter displays the following command usage for the pattern compositor:

```
usage: \XTOS\\release_number\bin\CompositeLink.exe [options] <CompositeFile>
Required Arguments:
    <CompositeFile> Name of the Composite file, which is to be parsed
Optional Arguments:
    --help         Display help message
    --noxml        No Composite XML file is needed, a single .pof file has to
    be given as input
```

The CompositeLink application (the compositor) parses the input XML block containing the details on patterns to be composited and creates a composite pattern object file (composite_domain_name.cpof) containing the shared data by the patterns in the composite.

In the multiple time domain applications, multiple composite object files are generated, one for each time domain.

5.3.3 XTOS Pattern CompositeLink Command Line Interface—Solaris

The command line interface for the XTOS Pattern compositor is as follows:

/XTOS/*release_number*/bin/CompositeLink.exe

Typing the above command with the --help parameter displays the following command usage for the pattern compositor:

```
usage: /XTOS/release_number/bin/CompositeLink.exe [options] <CompositeFile>
Required Arguments:
    <CompositeFile> Name of the Composite file, which is to be parsed
Optional Arguments:
    --help         Display help message
    --noxml        No Composite XML file is needed, a single .pof file has to
    be given as input
```

The CompositeLink application (the compositor) parses the input XML block containing the details on patterns to be composited and creates a composite pattern object file (composite_domain_name.cpof) containing the shared data by the patterns in the composite.

In the multiple time domain applications, multiple composite object files are generated, one for each time domain.

Test Template for Pattern Loading 6

The XTOS software includes a test template to facilitate the loading of multiple pattern files into the tester local memory. This template loads all patterns listed in a PatternBurst. The following Java code is an example of the pattern loading test template:

```
/*************************************************************************\
* (c) Copyright 2002-2003 NPTest, Inc.                                    *
* This computer program includes Confidential, Proprietary Information    *
* and is a trade secret of NPTest. All use, disclosure, and/or            *
* reproduction is prohibited unless authorized in writing by an officer *
* of NPTest. All Rights Reserved.                                         *
\*************************************************************************/

package com.nptest.sapphire.test_templates.pattern_loader;

import java.io.*;
import Sapphire.CAP.*;
import Sapphire.ITA.*;
import com.nptest.sapphire.test_templates.*;

public class PatternLoader extends TestTemplateBase
{
    // These variables are initialized by the setParam methods

  protected PatternBlock      Pattern = null;// Temp for a while ??
  protected PatternBurstBlock PatternBurst = null;

    private ParameterInfo PatternBurstParamInfo = new
ParameterInfo("PatternBurst", ParameterInfo.OPTIONAL, "PatternBurst to load");
    private ParameterInfo PatternParamInfo = new ParameterInfo("Pattern",
ParameterInfo.OPTIONAL, "Pattern block to load ");

  private boolean pattern_loaded = false;// Temp for a while ??
  private boolean pattern_burst_loaded = false;
  private boolean all_pattern_bursts_loaded = false;
```

```
//--------------------------------------------------------------------------
    /** Validate method:
    *    Checks that all parameters are correct.
    */

//--------------------------------------------------------------------------

  public void validate() throws java.lang.Exception
  {
        // move the checking of the PatternBurst block down to execute()
  }



//--------------------------------------------------------------------------
    /** Execute method:
    *
    *    Loads either the specified PatternBurst block or all PatternBurst
blocks.
    */

//--------------------------------------------------------------------------

    public Sapphire.CAP.TestStatus[] execute() throws java.lang.Exception
    {
        if ((Pattern != null) && !pattern_loaded)
        {
            Pattern.invoke("Load");
            System.out.println("Loaded pattern: " + Pattern.getS("Name"));
            pattern_loaded = true;
        }
        else if ((PatternBurst != null) && !pattern_burst_loaded)
        {
            PatternBurst.invoke("Load");
            System.out.println("Loaded PatternBurst: " +
PatternBurst.getS("Name"));
            pattern_burst_loaded = true;
        }
        else if (!all_pattern_bursts_loaded)
        {
            // Find all the PatternBurst blocks and load them.

            BlockManager block_manager = cap.getBlockManager("PatternBurst");
            String[] pattern_burst_block_names =
block_manager.getSarray("Blocks", 0, 0);

            if (pattern_burst_block_names.length == 0)
            {
                System.out.println("No PatternBurst blocks found in
testprogram");
            }

            for (int i = 0; i < pattern_burst_block_names.length; i++)
            {
```

```
                GenericBlock pattern_burst =
block_manager.getBlock(pattern_burst_block_names[i]);
                pattern_burst.invoke("Load");
                System.out.println("Loaded PatternBurst: " +
pattern_burst_block_names[i]);
            }

            all_pattern_bursts_loaded = true;
        }

        TestStatusArray testStatusArray = new TestStatusArray();
    return testStatusArray.get();
  }
}
```

Index

Numerics

96-pin DC Channel instrument
  pattern application 2-10
  timing block 2-10
  usage model 2-10

A

analog support
  capture signal 4-1
  example 4-1
  source signal 4-1
annotation
  statement 1-21
  STIL 1-21
  syntax 1-21
arguments 5-2, 5-4, 5-5
ASCII 1-10
ATPG tool 1-3, 3-1

B

block
  composite 5-6
  header 1-3
  MacroDefs 1-13
  NameMap 1-3, 5-2
  pattern 1-12, 1-14, 1-18
  PatternBurst 1-12, 1-18
  PatternExec 1-12, 1-18
  procedure 1-12, 1-13
  ScanChain 1-7
  ScanStructure 1-6, 3-2
  shift 3-1
  SignalGroups 1-5, 1-18
  signals 1-4, 1-18
  STIL 1-3
block (cont.)
  timing 1-7, 1-18, 2-10
  WaveformTable 1-7, 1-18
BNZ operation 1-17
BreakPoint statement 1-15, 2-7

C

call
  match 2-8
  statement 1-13, 1-14, 1-16, 2-8
  syntax 1-16
call until sync
  See CUS
comment
  statement 1-22
  syntax 1-22
compare state 1-9
compiler 5-7
CompileStil.exe 5-2
composite
  block 5-6
  file 5-8
  pattern
    creation 5-7
    image 5-6
CompositeLink
  application 5-8, 5-9
  utility 5-6
compositor 5-9
concatenation 1-4
condition
  statement 1-16
  syntax 1-16
construct 2-1

continuous loop 2-5
CUS 2-5, 2-8

D
DAP 2-4
DataBitCount 1-6
Dial a Pipe
  See DAP
domain 5-6
drive state 1-8

E
EDA
  ATPG tool 2-1
  environment 3-1
  tool 2-1
EESM 5-6
event sequence store memory
  See ESSM
expect state 1-9

F
F2
  data 1-12
  mode 1-6
F-data 1-6, 1-10
FM mode 1-10
FTest 1-12

H
header
  block 1-3
  statements 1-3

I
IEEE
  1450 ix, 1-1, 2-1
  1450.3 1-1
  1999 1-1, 2-1
IfNeed option 1-4
include
  file 1-4, 1-19
  statement 1-4
indeterminate state 1-9

K
KeepAlive vector
  statement 2-8
  syntax 2-8

L
label statement 1-22
latency pipe 2-3
loop
  continuous 2-5
  statement 1-17
  syntax 1-18

M
macro
  MacroDef 1-17
  statement 1-14, 1-17, 2-1
  syntax 1-17
MacroDef 1-17
MacroDefs
  block 1-13
  syntax 1-14
match call 2-8
MatchLoop
  statement 1-18, 2-7
  syntax 1-18
memory
  main 1-17, 2-9
  subroutine 1-17, 2-1, 2-2, 2-9
merge waveform 1-10

N
NameMap
  block 1-3, 5-2
  file 5-2
NPTAlternativeDataSourceDef 2-9
NPTAnalogCapture 4-1
NPTAnalogSource 4-1
NPTest UserKeywords
  NPTAlternativeDataSource 2-9
  NPTAnalogCapture 4-1
  NPTAnalogSource 4-1
  NPTIncMain 2-9
  NPTOpcode 2-1
  NPTSubroutine 1-12, 2-1, 2-8
  NPTWaitTrigger 2-6, 2-8
NPTIncMain 2-9
NPTIncMain statement 2-9
NPTOpcode 2-1
NPTSubroutine 1-12, 2-1, 2-8
NPTWaitTrigger 2-6, 2-8
null statement 2-7

O
opcode
  NPTOpcode 2-1
  PARAM 2-1
operation
  BNZ 1-17
  RPT 1-17

P
PARAM opcode 2-1
parameter passing 2-1, 2-9
PatList 1-12
pattern
  block 1-12, 1-14, 1-18
  compiler 5-2, 5-7
  compositor 5-9
  statement
    BreakPoint statement 1-15
    call 1-14, 1-16
    condition 1-14, 1-16
    GoTo 1-15
    loop 1-14, 1-17
    macro 1-14, 1-17
    MatchLoop 1-15, 1-18
    vector statement 1-14, 1-15
    WaveformTable reference 1-14, 1-15
  STIL 1-13
  subroutine support 2-1
PatternBurst
  block 1-12, 1-18, 5-6
  patterns 6-1
  syntax 1-12
PatternExec
  block 1-12, 1-18
  syntax 1-12
PatternLists 5-6
procedure
  block 1-12, 1-13
  construct 2-1
  statement 2-1
  syntax 1-13

Q
quotation marks 1-4

R
remap tables 5-6
repeat until match statement 2-7
resource tag
  period_name 1-8
  sequence_name 1-8
RPT operation 1-17
RPTM 2-7

S
scan
  chain 1-6, 1-7
  input 1-6
  output 1-6
  pattern 3-1
  vector 2-1
ScanChain block 1-7
ScanLength statement 1-7
ScanOutLength statement 1-7
ScanStructure
  block 1-6, 3-2
  syntax 1-6
SDS 5-6
sections 5-6
select sync type
  See SST
shared resources 5-6
shift
  block 3-1
  statement 3-1
signal
  declaration 3-1
  name 1-4
  state 1-13
  type 1-4
signal state 1-14
SignalGroups
  block 1-5, 1-18
  declaration 3-1
  example 1-5
  global 1-5
  syntax 1-5
signals
  block 1-4, 1-18
  syntax 1-4
source data select
  See SDS
source syntax ix
SST 5-6
standard test interface language
  See STIL

state
compare 1-9
drive 1-8
expect 1-9
indeterminate 1-9
signal 1-13, 1-14
statement
annotation 1-21
BreakPoint 1-15, 2-7
call 1-13, 1-14, 1-16, 2-8
comment 1-22
condition 1-16
header 1-3
include 1-4
KeepAlive vector 2-8
label 1-22
loop 1-17
macro 1-14, 1-17, 2-1
MatchLoop 1-18, 2-7
NPTIncMain 2-9
null 2-7
pattern 1-13, 1-14, 1-22, 2-1
procedure 2-1
ScanLength 1-7
ScanOutLength 1-7
shift 3-1
STIL 1-3
subroutine 2-2
UserKeywords 1-22
vector 1-15, 2-7
WaveformTable reference 1-15
STIL
annotation 1-21
block 1-3
comment 1-22
defination ix
extension 2-4
extension syntax 4-1
standards 1-1
statement 1-3
UserKeywords 1-22
subroutine
EDA compatible 2-2
extension 2-1
parameter passing 2-9
resource 2-1
statement 2-2
syntax 2-1, 2-2
SUBROUTINE_F 5-6
sync trigger pin 1-4
syntax
annotation statement 1-21
call statement 1-16
comment statement 1-22
condition statement 1-16
KeepAlive vector statement 2-8
loop statement 1-18
macro statement 1-17
MacroDefs block 1-14
MatchLoop statement 1-18
PatternBurst block 1-12
PatternExec block 1-12
procedure block 1-13
ScanStructure block 1-6
SignalGroups block 1-5
signals block 1-4
source 1-ix
subroutine memory 2-1, 2-2
timing block 1-7
vector statement 1-15

T

template
FTest 1-12
timing
block 1-7, 1-18, 2-10
syntax 1-7
trigger 2-3

U

UserKeywords
extensions 1-3
NPTAlternativeDataSource 2-9
NPTAnalogCapture 4-1
NPTAnalogSource 4-1
NPTIncMain 2-9
NPTOpcode 2-1
NPTSubrountine 2-1, 2-8
NPTSubroutine 1-12
NPTWaitTrigger 2-6, 2-8
statement 1-22
STIL statement 1-3
subroutine support 1-12, 2-1

V

vector
file 1-1
statement 1-15, 2-7
syntax 1-15

W

wait until sync
See WUS
waveform character
See WFC
waveform event 1-11
WaveformTable
analog support 4-1
block 1-7, 1-11, 1-18
reference 1-14
reference statement 1-15
WFC
ASCII 1-10
conventions 1-10
event mapping 1-10
resource tag 1-8
signal state 1-14
signals 1-13
WUS 2-5, 2-7

Reader Comments

We would appreciate your taking the time to complete this form and return it to us. Your answers are a valuable aid in improving the quality, and usefulness, of our documentation. All responses are confidential and remain the property of NPTest.

Publication: **XTOS Pattern User Manual**

Part Number: **P/N 57010495 Rev 6** **Date: December 2003**

How do you use this publication? ☐ As an introduction ☐ As a reference ☐ As a training text ☐ Other: _____

☐ Yes ☐ No If a user guide, is it easy to understand? If a reference, is it easy to use? If not, please explain why.

☐ Yes ☐ No If a user guide, does it contain all the relevant information to understand and operate the product? If not, what additional information would you like?

☐ Yes ☐ No Were the illustrations helpful? If not, please explain why.

☐ Yes ☐ No Would you like more illustrations? If so, of which subjects?

☐ Yes ☐ No Is the information in the publication accurate? If not, please identify the inaccuracies.

Other Comments

Name Title Date

Company Division

Street City

State Zip or Country Phone

Return Address (USA)

Technical Publications Manager
NPTest
150 Baytech Drive
San Jose, CA 95134

Fold here

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7698 SAN JOSE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

NPTest
150 BAYTECH DRIVE
SAN JOSE, CA 95134
Attn.: Technical Publications Manager